

**UNIVERSIDADE FEDERAL DE SÃO CARLOS**

**CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA**

**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**IMPLEMENTAÇÃO DE UMA API PARA EMULAR  
O KERNEL DE TEMPO REAL RTXC NO LINUX/GNU  
VISANDO APLICAÇÕES MULTICORE**

**DANIEL SANT' ANNA CONSIGLIERI**

**ORIENTADOR: PROF. DR. MARCIO MERINO FERNANDES**

**CO-ORIENTADOR: PROF. DR. CÉLIO ESTEVAM MORON**

São Carlos - SP  
Fevereiro/2017

**UNIVERSIDADE FEDERAL DE SÃO CARLOS**

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**IMPLEMENTAÇÃO DE UMA API PARA EMULAR O  
KERNEL DE TEMPO REAL RTXC NO LINUX/GNU  
VISANDO APLICAÇÕES MULTICORE**

**DANIEL SANT' ANNA CONSIGLIERI**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Metodologias e Técnicas de Computação

Orientador: Prof. Dr. Marcio Merino Fernandes

São Carlos - SP  
Fevereiro/2017

# **UNIVERSIDADE FEDERAL DE SÃO CARLOS**

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

## **Implementação de uma API para Emular o Kernel de Tempo Real RTXC no Linux/GNU Visando Aplicações Multicore**

**DANIEL SANT' ANNA CONSIGLIERI**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Metodologias e Técnicas de Computação.

Aprovado em: 01 de Fevereiro de 2017.

Membros da Banca:

---

Prof. Dr. Marcio Merino Fernandes

(Orientador – DC-UFSCar)

---

Prof. Dr. Prof. Dr. Célio Estevan Morón

(Co-Orientador – DC-UFSCar)

---

Prof. Dr. Alexandro Baldassin

UNESP – Rio Claro

---

São Carlos - SP  
Fevereiro/2017



# UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia  
Programa de Pós-Graduação em Ciência da Computação

---

## Folha de Aprovação


---

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Daniel Sant Anna Consiglieri, realizada em 01/02/2017:




---

Prof. Dr. Márcio Merino Fernandes  
UFSCar



---

Prof. Dr. Célio Estevan Moron  
UFSCar



---

Prof. Dr. Alexandre José Baldassin  
UNESP

*Dedicatória*

*A Deus fonte de amor, justiça e sabedoria, em gratidão  
por fornecer saúde e meios para completar esta jornada.*

# AGRADECIMENTO

Ao meu pai Francisco Antonio Zanotto Consiglieri,  
pelo amor sem medida, por todo incentivo, apoio,  
paciência e conselhos fundamentais em minha formação  
A minha mãe Silvia Sant´Anna Consiglieri,  
Pela vida e por me ensinar bons valores.

Ao meu avô Zacarias Sant´Anna,  
por sempre me incentivar a aprender,  
dedicar nos estudos  
e ser um exemplo no saber

Aos meus irmãos  
Bárbara Sant´Anna Consiglieri Val Magalhães,  
Francisco Antonio Zanotto Consiglieri Junior e  
Renata Sant´Anna Consiglieri  
cunhado Eric Val Magalhães  
e sobrinho Nathan Consiglieri Val Magalhães  
por me ensinarem o significado de viver em família, amizade e apoio

A minha tia Eunice Margarida Consiglieri,  
por me incentivar, apoiar nos estudos, motivar a ser melhor  
e seus conselhos

Aos meus tios João Augusto de Sant´Anna Neto,  
Judite Mariano Sant´Anna,  
José Augusto Sant'Anna e Adriana Sant´Anna  
pelo incentivo e conselhos

Ao Prof. Dr. Marcio Merino Fernandes,  
por me ajudar a me tornar um pesquisador, por acreditar no meu potencial, pela paciência, pelos conselhos, por toda ajuda, conhecimento compartilhado e amizade.

Ao Prof. Dr. Célio Estevan Morón,  
Por sempre estar disposto a ajudar, sendo um pai na pesquisa, pela paciência, pelos conselhos, pela parceria e co-autoria no desenvolvimento da API-RXTC-Linux, pelo conhecimento compartilhado e amizade.

Ao Prof. Dr. Alexandro Baldassin,  
Pela participação como membro externo da banca examinadora, revisão da dissertação e proposta de correções.

Ao Prof. Dr. Hélio Crestana Guardia,  
Pela participação na defesa, ajuda na revisão da dissertação e proposta de correções.

Ao meu amigo Antonio Ideguchi  
por me ajudar e incentivar a me dedicar nos estudos

A todos professores do Departamento de Computação  
responsáveis por minha formação

A todos funcionários, colegas, amigos  
do Departamento de Computação e  
da UFSCar

Ao CNPq pelo apoio financeiro através da bolsa de mestrado

Agradecimento in memoriam  
à minha bisavó Eunice Figueira,  
às minhas avós Eunyce Zanotto Consiglieri,  
e Nélia Therezinha Martinelli Sant'Anna

a minha tia-avó Délia Figueira Zanotto,

Por sempre me ajudarem e torcerem por meu sucesso.



*Só é digno da liberdade, como da vida, aquele que se empenha em conquistá-la.*

*Johann Goethe*

# RESUMO

O advento dos processadores multicore representou um grande avanço no desempenho da computação paralela. Os sistemas embarcados seguindo essa tendência, melhoraram o poder computacional disponível para sistemas de tempo real, possibilitando a utilização da cadeia de desenvolvimento GNU/Linux. Consolidado a mais de vinte anos no mercado, o RTOS RTX da Quadros Systems, foi escolhido para ser emulado no ambiente Linux. Com o objetivo de se emular o kernel RTX, a API-RTX-Linux foi escrita em C e obteve resultados determinísticos além de distribuição homogênea para tempo de resposta das chamadas implementadas, sendo possível seu uso para aplicações de tempo real.

**Palavras-chave:** emulador; sistema de tempo-real; sistemas embarcados; kernel RTX; multicore;

# ABSTRACT

The advent of multicore processors represented a major advance in the performance of parallel computing. Embedded systems following this trend have improved the computing power available for real-time systems, enabling the GNU/Linux development chain. Consolidated to more than twenty years in the Linux environment. In order to emulate the RTX kernel, an API-RTX-Linux was written in C and obtained deterministic results as well as homogeneous distribution for the response time of implemented calls, being possible its use for real-time applications.

**Keywords:** emulator; real-time system; embedded systems; RTX; kernel RTX; multicore;

# LISTA DE FIGURAS

Figura 2.1 - Representação arquitetura memória de acesso uniforme.....	19
Figura 2.2 - Representação arquitetura memória de acesso não uniforme.....	20
Figura 2.3 - Representação da arquitetura de memória distribuída. ....	21
Figura 2.4 - O STR reage a entrada do ambiente produzindo uma saída que afeta o ambiente[Adaptada de (LABROSSE et al., 2008)] .....	22
Figura 2.5 - Organização lógica do Intel Core Duo. [Adaptado de:(LIN; SNYDER, 2009)] .....	26
Figura 2.6 - Organização lógica do AMD Dual Core Opteron.[Adaptado de: (LIN; SNYDER, 2009)] .....	27
Figura 2.7 - Diagrama lógico de Multiprocessadores Simétricos. [Adaptada de: (LIN; SNYDER, 2009)] .....	29
Figura 2.8 - Arquitetura processador Cell.....	31
Figura 2.9 - À esquerda um processo com um thread e a direita um processo <i>multithreaded</i> .....	32
Figura 2.10 - Esquema do modelo de passagem de mensagens .....	34
Figura 2.11 - Esquema de operação PGAS – [Adaptado de: (BARNEY, 2015)].....	36
Figura 2.12 - Ferramenta TotalView analisando código MPI [Extraído de: (BARNEY,2015) ] .....	40
Figura 2.13 - Ferramenta Allinea DDT analisando código [Extraída de (“Allinea DDT”) ] .....	41
Figura 2.14 - Ferramenta Intel Parallel Studio XE [Extraída de (“Intel® Parallel Studio XE 2016”) ] .....	42
Figura 3.1 - Processador 8 núcleos Sandy Bridge-EP [Extraído de: (MOLKA et al., 2014)] .....	44
Figura 3.2 - Processador 16 núcleos Bulldozer. [Extraído de (MOLKA et al., 2014)]. .....	44
Figura 3.3 – Estrutura e componentes do Cortex – A72 [Extraído de: (“Cortex-A72 Processor - ARM”)].....	45
Figura 3.4 - Configuração Xenomai dual de kernel [Adaptada de: (“Start Here – Xenomai”)].....	49
Figura 3.5 - Configuração de kernel unitária Xenomai. [Adaptada de: (“Start Here – Xenomai”)].....	50
Figura 3.6 - Diagrama de blocos da arquitetura SimpleScalar [adaptado de: (AUSTIN et al., 2002)] .....	51

Figura 3.7 - Arquitetura da solução StarHPC [Adaptado de (IVICA et al., 2009)].....	52
Figura 4.1 - Zonas de Operação Quadros RTX [Adaptado de: (“RTXC Quadros Manuals”)] .....	54
Figura 4.2 - Esquema dos componentes RTX/ms e RTX/ss [Adaptado de: (“RTXC Quadros Manuals”)].....	55
Figura 4.3 - Exemplo de tabela de escalonamento [Adaptado de: (“RTXC Quadros Manuals”)] .....	57
Figura 4.4 - Primeiro exemplo de sequência de tempo Round-Robin. [Adaptado de: (“RTXC Quadros Manuals”)].....	58
Figura 4.5 - Segundo exemplo de sequência de tempo Round-Robin. [Adaptado de: (“RTXC Quadros Manuals”)].....	58
Figura 4.6 - Exemplo de escalonamento por prioridade. [Adaptado de: (“RTXC Quadros Manuals”)].....	59
Figura 4.7 - Hierarquia do gerenciamento de eventos. [Extraído de: (“RTXC Quadros Manuals”)] .....	65
Figura 4.8 - Mostra alarme sendo armado com valor de disparo inicial de $C_1 - C_0$ e valor disparo reciclado $C_{n+2} - C_{n+1}$ , para $0 \leq n \leq 2$ . [Adaptado de: (“RTXC Quadros Manuals”)].....	67
Figura 4.9 - Alarme não-cíclico, armado com valor inicial de contador $C_0$ e disparando em $C_1$ . [Adaptado de: (“RTXC Quadros Manuals”)] .....	67
Figura 4.10 - Duração de alarme não cíclico de tempo de duração um tick, situação 1. [Adaptado de: (“RTXC Quadros Manuals”)] .....	68
Figura 4.11 - Duração de alarme não cíclico de tempo de duração um tick, situação 2. [Adaptado de: (“RTXC Quadros Manuals”)] .....	68
Figura 4.12 - Operações básicas do Pipe [Adaptado de: (“RTXC Quadros Manuals”)]. .....	71
Figura 5.1 - Modelo de Kernel Dual.....	73
Figura 5.2 - Modelo de execução espaço de usuário.....	74
Figura 5.3 - Modelo de alteração interna do <i>kernel</i> .....	74
Figura 5.4 - Pseudocódigo da chamada <code>XX_ScheduleThread</code> .....	75
Figura 5.5 - Pseudocódigo de <code>KS_SuspendTask</code> .....	77
Figura 5.6 - Pseudocódigo de <code>XX_ResumeTask</code> .....	77
Figura 5.7 - Pseudocódigo da chamada <code>XX_ProcessEventSourceTick</code> . .....	78
Figura 5.8 - Pseudocódigo de escalonador de disparos do Alarme. ....	80
Figura 5.9 - Pseudocódigo de função <code>KS_TestAlarmW</code> . ....	81

Figura 5.10 - Pseudocódigo da chamada KS_PutQueueDataW.....	82
Figura 5.11 - Pseudocódigo da chamada KS_GetQueueDataW.....	82
Figura 5.12 - Pseudocódigo do serviço KS_SendMsgW.....	84
Figura 5.13 - Pseudocódigo de chamada KS_AckMsg. ....	84
Figura 5.14 - Pseudocódigo de ReceiveMsgW. ....	84
Figura 5.15 - Pseudocódigo gerenciador de filas de espera Partição de Memória. ..	86
Figura 5.16 - Pseudocódigo do serviço KS_AllocBlkW. ....	86
Figura 5.17 - Pseudocódigo da função KS_FreeBlk.....	87
Figura 6.1 - Pseudocódigo para o benchmark de preempção de tarefa.....	91
Figura 6.2 - Resultados para o benchmark de preempção de tarefa. ....	92
Figura 6.3 - Pseudocódigo para o benchmark da participação de memória. ....	93
Figura 6.4 - Gráficos de resultados requisição/devolução bloco de memória em Partição. ....	93
Figura 6.5 - Pseudocódigo para o benchmark do mailbox bloqueante.....	94
Figura 6.6 - Resultados para o benchmark de Mailbox bloqueante. ....	96
Figura 6.7 - Resultados para o benchmark de mailbox não bloqueante. ....	97
Figura 6.8 - Pseudocódigo para o benchmark do mailbox não bloqueante.....	97
Figura 6.9 - Pseudocódigo para o benchmark da fila bloqueante. ....	98
Figura 6.10 - Resultados para o benchmark da fila.....	99
Figura 6.11 - Pseudocódigo para benchmark do alarme.....	100
Figura 6.12 - Gráficos do benchmark para disparo de alarme. ....	101
Figura 6.13 - Pseudocódigo para o benchmark do mutex. ....	101
Figura 6.14 - Gráficos do benchmarch para mutex bloqueante.....	102
Figura 6.15 - Pseudocódigo para o benchmark de semáforo.....	103
Figura 6.16 - Resultados para o benchmark de semáforo. ....	103
Figura 6.17 - Mandelbrot obtido no benchmark.....	105

# LISTA DE TABELAS

Tabela 2.1 - Comparação entre Sistemas de Tempo Compartilhado e de Tempo Real. [Adaptado de: (LABROSSE et al., 2008)].....	24
Tabela 5.1 - Outras chamadas do objeto Thread .....	76
Tabela 5.2 - Outras chamadas para objeto Tarefa.....	77
Tabela 5.3 - Outras chamadas de <i>Event Source</i> .....	78
Tabela 5.4 - Outras chamadas de Contador. ....	79
Tabela 5.5 - Outras chamadas objeto Alarme. ....	81
Tabela 5.6 - Outras chamadas do objeto Fila.....	83
Tabela 5.7 - Outras chamadas do objeto Mensagem.....	85
Tabela 5.8 - Outras chamadas do objeto Maibox.....	85
Tabela 5.9 - Outras chamadas do objeto Partição. ....	87
Tabela 5.10 - Chamadas referentes ao objeto Exceção. ....	87
Tabela 5.11 - Chamadas referentes ao objeto Mutex. ....	88
Tabela 5.12 - Chamadas referentes ao objeto Semáforo.....	89
Tabela 6.1 - Descrição dos sistemas em que os benchmarks e testes foram realizados .....	90
Tabela 6.2 - Coeficiente de variação de preempção de tarefa.....	92
Tabela 6.3 - Coeficiente de variação de requisição/devolução bloco de memória em Partição. ....	94
Tabela 6.4 - Coeficiente de variação para Mailbox bloqueante (Um núcleo). ....	95
Tabela 6.5 - Coeficiente de variação para Mailbox bloqueante (Dois núcleos). ....	95
Tabela 6.6 - Coeficiente de variação para o benchmark de mailbox não bloqueante. .....	97
Tabela 6.7 - Coeficiente de variação para o benchmark da fila.....	99
Tabela 6.8 - Coeficiente de variação para benchmark do alarme. ....	100
Tabela 6.9 - Coeficiente de variação para mutex bloqueante. ....	102
Tabela 6.10 - Coeficiente de variação para o benchmark de semáforo. ....	103
Tabela 6.11 - Coeficiente de Variação das medidas do Benchmak .....	104
Tabela 6.12 - Tempo para o teste Mandelbrot .....	106
Tabela 6.13 - Ganho no teste Mandelbrot com duas Tarefas e dois cores.....	106





# LISTA DE ABREVIATURAS E SIGLAS

- ARM** – *Advanced RISC Machine.*
- CFS** – *Completely Fair Scheduler.*
- CPU** – *Central Processing Unity*
- CUDA** – *Compute Unified Device Architecture.*
- FIFO** – *First in, first out.*
- FPGA** – *Field-Programmable Gate Array.*
- FPU** – *Floating-point unit.*
- GPU** – *Graphics Processing Unit.*
- IEEE** – *Institute of Electrical and Electronics Engineers.*
- ISO** – *International Organization for Standardization.*
- ISR** – *Interrupt Service Routine.*
- IPC** – *Inter-process Communication.*
- HPC** – *High Performance Computing.*
- LIFO** – *Last in, first out.*
- LWP** – *Lightweight Processes.*
- LSB** – *Least Significant Bit.*
- MESI** – *Modified, Exclusive, Shared and Invalid*
- MOESI** – *Modified, Owned, Exclusive, Shared and Invalid.*
- MSB** – *Most Significant Bit.*
- NUMA** – *Non-Uniform Memory Access.*
- OpenCL** – *Open Computing Language.*
- PGA** – *Partitioned Global Address Space.*
- POSIX** – *Portable Operating System Interface.*
- QPI** – *Quick Path Interconnect.*
- RPC** – *Remote Procedure Call.*
- RTOS** – *Real-Time Operating System.*
- RTXC** – *Real-Time Executive in C.*
- SMP** – *Symmetric Multiprocessor.*
- SoC** – *System on Chip.*
- STR** – *Sistemas de Tempo Real.*

**SPE** – *Synergistic Processing Elements.*

**SRI** – *System Request Interface.*

**UMA** – *Uniform Memory Access.*

**VFP** – *Vector Floating-Point.*

# SUMÁRIO

<b>CAPÍTULO 1 - INTRODUÇÃO.....</b>	<b>13</b>
1.1 Taxonomia dos sistemas computacionais .....	14
1.2 Desenvolvimento paralelo .....	14
1.3 Objetivos gerais do trabalho .....	16
1.4 Organização do trabalho .....	17
<b>CAPÍTULO 2 - CONCEITOS BÁSICOS .....</b>	<b>18</b>
2.1 Conceitos básicos .....	18
2.1.1 Memória compartilhada .....	19
2.1.1.1 Memória de Acesso Uniforme (UMA -Uniform Memory Access) .....	19
2.1.1.2 Memória De Acesso Não Uniforme (NUMA - Non-Uniform Memory Access).....	20
2.1.2 Memória distribuída .....	21
2.2 Sistemas Operacionais De Tempo Real .....	22
2.2.1 Diferenças entre Sistema Operacional de Tempo Real e Sistema de Tempo compartilhado.....	23
2.3 Arquiteturas Multicores.....	24
2.3.1 Motivação para Desenvolvimento .....	24
2.3.2 Hardware .....	25
2.3.2.1 Intel Core Duo .....	25
2.3.2.2 AMD Dual Core Opteron .....	26
2.3.2.3 Comparação entre Intel e AMD .....	28
2.3.2.4 Arquiteturas de Multiprocessadores Simétricos .....	28
2.3.2.5 Arquitetura de chips heterogêneos.....	30
2.3.3 Modelos de Programação Paralela .....	32
2.3.3.1 Modelo de Threads.....	32
2.3.3.2 Modelo de Passagem de mensagens .....	34
2.3.3.3 Modelo PGAS.....	35
2.4 Mecanismos de comunicação entre processos.....	37
2.4.1 Comunicação cliente servidor .....	38
2.5 Ferramentas para Programação paralela.....	39

<b>CAPÍTULO 3 - REVISÃO BIBLIOGRÁFICA .....</b>	<b>43</b>
3.1 Arquiteturas multicore recentes.....	43
3.2 Arquitetura multicore de sistema embarcado recente .....	45
3.3 Métodos para programação paralela em multicores .....	46
3.4 Ferramentas para programação paralela em multicores de tempo real .....	48
3.5 Ferramentas educacionais para programação paralela .....	50
<b>CAPÍTULO 4 - DISCUSSÃO DOS OBJETOS DO KERNEL RTXC .....</b>	<b>53</b>
4.1 Responsabilidades de um kernel de tempo real.....	53
4.2 Introdução ao kernel RTXC .....	53
4.3 Níveis ( <i>Levels</i> ).....	55
4.4 Objetos executáveis do <i>Kernel</i> RTXC .....	56
4.4.1 Exceção (Exception) .....	56
4.4.2 Threads .....	56
4.4.3 Tarefa ( <i>Task</i> ).....	60
4.5 Semáforo.....	62
4.6 Mutex.....	63
4.7 Partição de Memória .....	63
4.8 Fonte de Evento ( <i>Event Source</i> ) .....	64
4.9 Contador ( <i>Counter</i> ) .....	66
4.10 Alarm (Alarme) .....	66
4.11 Fila ( <i>Queue</i> ) .....	69
4.12 <i>Mailbox</i> .....	69
4.13 Mensagem (Message).....	70
4.14 Pipe .....	71
<b>CAPÍTULO 5 - IMPLEMENTAÇÃO DA API .....</b>	<b>72</b>
5.1 Considerações iniciais.....	72
5.2 Linux em Tempo Real .....	73
5.2.1 Implementação de kernel extra .....	73
5.2.2 Evitar usar o <i>kernel</i> para requisitos de tempo real .....	74
5.2.3 Modificar características do <i>kernel</i> do Linux.....	74
5.3 Desenvolvimento API .....	75
5.3.1 Thread.....	75

5.3.2 Tarefa .....	76
5.3.3 Fonte de eventos, Contador e Alarme .....	78
5.3.4 Fila (Queue) .....	82
5.3.5 Mailbox e Mensagens .....	83
5.3.6 Partição de Memória .....	85
5.3.7 Exceção .....	87
5.3.8 Mutex .....	88
5.3.9 Semáforo .....	88
<b>CAPÍTULO 6 - BENCHMARKS E TESTE NA API-RTXC-LINUX .....</b>	<b>90</b>
6.1 Introdução benchmark .....	90
6.1.1 Preempção de tarefa .....	91
6.1.2 Requisição/devolução de blocos em partição de memória .....	92
6.1.3 Mailbox .....	94
6.1.4 Fila ( <i>Queue</i> ) .....	98
6.1.5 Alarme .....	99
6.1.6 Mutex .....	101
6.1.7 Semáforo .....	102
6.1.8 Resumo dos resultados benchmark .....	104
6.2 Testando a API com Mandelbrot Multicore .....	105
<b>CAPÍTULO 7 - CONCLUSÃO .....</b>	<b>107</b>
7.1 Uso da API-Linux-RTXC .....	108
7.2 Contribuições e Limitações .....	109
7.3 Trabalhos Futuros .....	110
<b>REFERÊNCIAS .....</b>	<b>111</b>
<b>APÊNDICE A .....</b>	<b>114</b>

# Capítulo 1

## INTRODUÇÃO

---

Buscando uma constante evolução, processadores e a arquitetura de computadores foram aprimorados durante décadas desde seu surgimento e continuam até hoje. No princípio houve uma tendência de se aumentar a frequência de trabalho desses processadores, uma vez que a frequência está diretamente relacionada ao número de instruções executadas num período de tempo.

A abordagem de aumento de *clock* caracterizava em um “ganho fácil de desempenho” para os programadores, uma vez que a melhoria no desempenho dos sistemas computacionais pesava mais para os requisitos de hardware do que com a melhoria de técnicas de programação (LIN; SNYDER, 2009).

No entanto, com relação ao hardware, o aumento de *clock* significa também o aumento de dissipação térmica, ficando evidente para os fabricantes que essa característica se tornaria uma limitação com o passar do tempo. A partir daí, surge a ideia dos computadores multicore, ou seja, em um mesmo chip são condensados dois ou mais núcleos.

A proposição da lei de Moore que prevê a cada dois anos a duplicação do número de transistores, e conseqüentemente melhorias no processador passaram a ditar a política dos fabricantes de hardware, que puderam adicionar novos recursos e elementos a arquitetura melhorando ainda mais o desempenho dos processadores.

A introdução de computadores multicore possibilitou que outros paradigmas de programação fossem utilizados, pois a utilização desses computadores em aplicações anteriormente escritas de forma serial não representava um ganho de desempenho. Era necessário então surgimento de modelos e ferramentas de programação paralela.

## 1.1 Taxonomia dos sistemas computacionais

Proposta por Flynn, a classificação de sistemas computacionais engloba as principais organizações de computadores existentes agrupadas em quatro categorias (FLYNN, 1972):

- SISD (*Single Instruction, Single Data*): Esse tipo de computador é puramente serial, apenas uma instrução é executada por vez e apenas um dado é processado. O tipo mais simples e mais antigo computador representa essa categoria

- SIMD (*Single Instruction, Multiple Data*): É um tipo de computador paralelo, em que todas as unidades de processamento executam a mesma instrução em diferentes dados. Essa arquitetura é utilizada em processadores vetoriais e GPUs.

- MISD (*Multiple Instruction, Single Data*): Nesse tipo de computador um único fluxo de dados alimenta diversas unidades de processamento executando diferentes instruções sobre esse mesmo dado. Essa arquitetura embora não usual, é importante para sistemas redundantes e com baixa tolerância a falhas.

- MIMD (*Multiple Instruction, Multiple Data*): nesse tipo de computador paralelo, diversas instruções são executadas usando diferentes fluxos de dados. Computadores multicore e sistemas distribuídos são exemplos dessa arquitetura.

## 1.2 Desenvolvimento paralelo

No paradigma de programação sequencial, um conjunto de instruções é executado de forma serial instrução por instrução. Na programação paralela, diversas operações são executadas simultaneamente.

Com relação à paralelização em nível de hardware pode-se citar: ILP (*Instruction Level Parallelism*) que é a forma de paralelização no nível de instrução (RAUBER; RUNGER, 2012). Nessa abordagem, uma instrução é subdividida em quatro sub-operações: carregamento, decodificação, execução e escrita. Com essa mudança, não é necessário terminar o processamento de uma instrução para começar

a carregar e executar as sub-operações da próxima instrução, permitindo que mais de uma instrução seja executada simultaneamente. Esse procedimento, conhecido também como *pipelining* de instruções, associado a predições de desvio é responsável pelo aumento no desempenho de um processador.

Outra forma de paralelismo para hardware foi a *Simultaneous Multithreading* (SMT) solução lançada pela Intel também chamada por Hyper-Threading. Nessa tecnologia um núcleo físico executa dois núcleos lógicos, cada processador lógico mantém as suas próprias configurações de arquiteturas de estado, compartilhando apenas os recursos de execução. Essa abordagem permite ganho no desempenho executando mais threads, fazendo poucas modificações no tamanho do desenho do processador (KOUFATY; MARR, 2003).

A programação paralela abriu portas para que se explorasse a capacidade disponibilizada pelos multicores. Porém, sua implementação não é trivial e nem aplicável 100% a todos os contextos.

De acordo com a lei de Amdahl, que calcula o ganho de desempenho em aplicações paralelas, a variável  $f$  corresponde à porcentagem do código que precisa ser executado sequencialmente, assumindo valores  $0 \leq f \leq 1$ , e  $p$  número de processadores do computador paralelo. O ganho (*speedup*) máximo é representado pela inequação  $\psi$  (QUINN, 2004):

$$\psi \leq \frac{1}{f + (1-f)/p}$$

Conforme a inequação apresentada, o ganho pode ser igual ou menor que o lado direito da expressão, uma vez que gastos com sincronizações e comunicação entre os componentes não são computados nessa inequação o ganho tende a ser ligeiramente menor.

Como principais modelos de programação paralela em multicores podemos citar: modelo de memória compartilhada, passagem de mensagens e modelo de dados paralelos.

No modelo de memória compartilhada, o paralelismo ocorre através de posições de memória compartilhadas entre os diversos processadores através de um mesmo barramento local que conecta processadores a memória. Diversas tarefas são realizadas simultaneamente tirando proveito dos vários núcleos se comunicando e sincronizando através dessa área compartilhada.



No modelo de passagem de mensagens a organização, paralelização e divisão das tarefas ocorre por meio de trocas de mensagens. Esse modelo favorece a utilização de sistemas distribuídos, embora possa ser utilizado com arquiteturas de memória compartilhada.

No modelo de dados paralelos, também conhecido como PGAS (*Partitioned Global Address Space*), temos um espaço de endereço global. Grupos de tarefas trabalham coletivamente executando uma mesma operação em diferentes áreas de um mesmo arranjo de dados.

### 1.3 Objetivos gerais do trabalho

As arquiteturas multicores possibilitaram a utilização de tecnologias de processamento paralelo para uso computacional. Ao se programar utilizando o paradigma paralelo é possível obter ganho em desempenho se comparado ao paradigma de programação sequencial.

A complexidade de programação paralela é maior que a programação sequencial, isso se deve preocupações adicionais tais como: sincronização de recursos compartilhados, operações de coerência, existência de regiões críticas e barreiras.

Para sistemas embarcados, que são sistemas de computador construídos dentro de um dispositivo e funcionando como parte do mesmo (BERGER, 2002) existem restrições adicionais. Um sistema embarcado crítico precisa gerar saídas respeitando a intervalos pré-definidos de tempo, podendo ser classificado: “*soft*” em que falhas correspondem a degradação de desempenho, ou “*hard*” cujas falhas comprometem o resultado (LABROSSE et al., 2008).

O objetivo deste projeto é prover uma API que permita, aos desenvolvedores que utilizam o tradicional RTOS (*RealTime Operating Systems*), portarem suas aplicações para o ambiente de desenvolvimento baseado no Linux e mostrar através de testes e benchmark o quanto pode ser adequada a API para o contexto que foi idealizada.

## 1.4 Organização do trabalho

Este trabalho está organizado da seguinte forma:

No capítulo 2 apresentam-se a revisão teórica de conceitos de arquiteturas multicore, paradigmas paralelos, sistemas de tempo real.

No capítulo 3 tem-se a revisão bibliográfica das arquiteturas atuais, ferramentas de programação paralela e educacional.

No capítulo 4 encontra-se a discussão dos objetos do kernel RTX.

No capítulo 5 são discutidas as estratégias para se tornar o Linux em tempo real, apresentada as chamadas que a API contém e os detalhes de como as principais chamadas foram implementadas.

No capítulo 6 apresentam-se os resultados do benchmark aplicado na API desenvolvida e o teste Mandelbrot Multicore.

No capítulo 7 está a conclusão do trabalho, apresentando-se as contribuições e limitações do trabalho desenvolvido e ideias para trabalhos futuros.

Nos apêndices encontram-se os resultados de benchmark que não foram colocados no capítulo 6.

# Capítulo 2

## CONCEITOS BÁSICOS

---

### 2.1 Conceitos básicos

No início da computação os programas eram escritos de forma sequencial. Cada instrução era executada uma a uma. Cientistas na busca de se construir máquinas mais velozes, desenvolveram novas tecnologias e processadores que atingiam *clocks* de frequências cada vez mais altas.

O aumento de *clock* tinha relação direta com aumento da vazão, ou seja, como a instrução estava atrelada a ciclos de *clock*, o aumento de ciclos por período significava um aumento de instruções que poderiam ser executadas num determinado período de tempo. Entretanto o aumento na frequência de *clock* também correspondia a um aumento na dissipação térmica. Passado algum tempo o aumento de dissipação térmica se tornou inviável devido à grande dissipação de calor por núcleo.

A crescente demanda por sistemas computacionais mais eficientes abriu a possibilidade de um outro campo, a computação paralela. Desta forma não seria mais o aumento do *clock* o único responsável pelo aumento da vazão e de tarefas executadas, mas o aumento de componentes de processamento (pipelines), núcleos e unidades de processamentos interligadas entre si.

## 2.1.1 Memória compartilhada

Na memória compartilhada, os núcleos de processamento têm acesso de toda memória através de um espaço de endereço global. Com isso, diversos núcleos podem operar de maneira independente utilizando esses espaços de memória compartilhados. Mudanças nos valores de memória são visíveis a todos os processadores. As máquinas de memória compartilhada têm sido classificadas em (Barney,2015):

### 2.1.1.1 Memória de Acesso Uniforme (UMA -Uniform Memory Access)

Sendo muito representada nos dias atuais através das SMP (*Symmetric Multiprocessor Machines* – Máquinas de multiprocessador simétrico). Nessas máquinas processadores idênticos tem acesso igual e de mesmo tempo as porções de memória. Algumas vezes chamado de CC-UMA (Cache Coerente UMA), isso significa que atualizações feitas na memória passam a ser conhecidas por todos os processadores. A coerência de cache é realizada em nível de hardware. Na *Figura 2.1* temos a representação da arquitetura UMA.

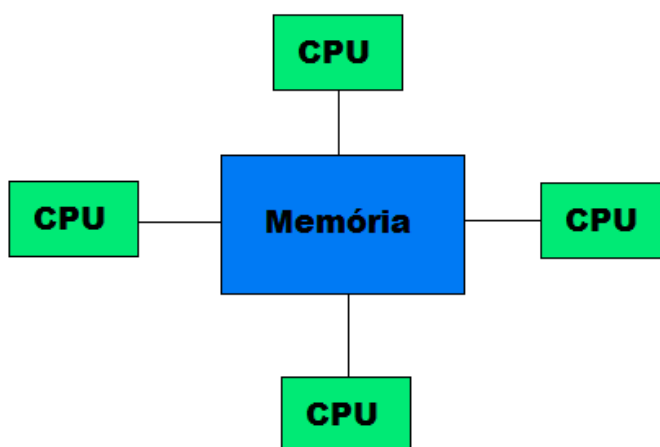


Figura 2.1 - Representação arquitetura memória de acesso uniforme

### 2.1.1.2 Memória De Acesso Não Uniforme (NUMA - Non-Uniform Memory Access)

São computadores geralmente construídos pela junção física de duas ou mais SMP. Uma SMP pode acessar diretamente a memória de outra SMP. Nem todos os processadores tem acesso igual e mesmo tempo de acesso a toda a memória. Acesso através do link das memórias é lento. Caso exista coerência de cache é então chamado de CC-NUMA (Cache Coerente NUMA). A *Figura 2.2* mostra um exemplo de NUMA:

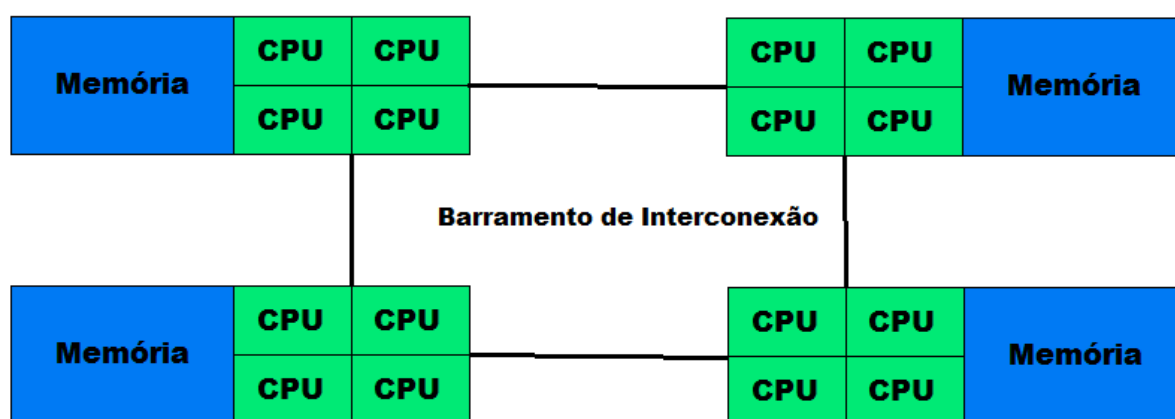


Figura 2.2 - Representação arquitetura memória de acesso não uniforme.

Como vantagens, a memória compartilhada fornece facilidades na programação graças a perspectiva de acesso global às posições de memória. Podemos citar também a uniformidade e velocidade de troca de mensagens devido à proximidade dos núcleos e memória.

Como desvantagens pode-se destacar a ausência de escalabilidade entre CPU e memória. Adicionando mais núcleos aumenta-se geometricamente o tráfego de informação no barramento memória-CPU, além de se aumentar a complexidade no gerenciamento da coerência de caches. Outra desvantagem é que o programador além de ser responsável pela sincronização é também responsável pelo acesso correto à memória global.

### 2.1.2 Memória distribuída

A tecnologia envolvida na construção de sistemas de memória distribuída pode variar bastante. Porém uma coisa é comum: o acesso de memórias distribuídas é intermediado por uma rede que conecta as memórias entre processadores. O mapeamento das memórias não é global, como no caso da memória compartilhada. Portanto, como o mapeamento passa a ser local não existe a necessidade de se verificar a coerência de cache. Por outro lado, em geral, a sincronização e comunicação devem ser explicitamente definidas pelo programador. A construção da rede de comunicação pode variar sendo desde a redes de altíssimo desempenho a redes comuns como a Ethernet. Na Figura 2.3 temos um esquema de memória distribuída:

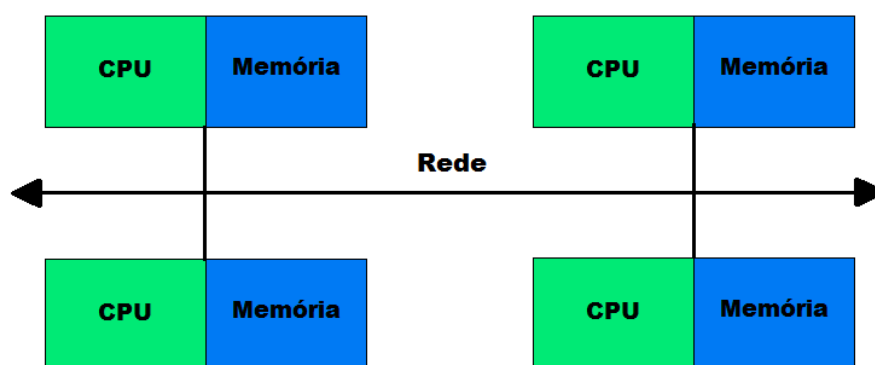


Figura 2.3 - Representação da arquitetura de memória distribuída.

Como vantagens podemos citar:

- A memória é escalável com o número de processadores, ou seja, o aumento do número de processadores também aumenta o tamanho de memória proporcionalmente.
- Cada processador pode acessar rapidamente sua própria memória, sem congestionamento ou atrasos ocorridos pelas políticas de coerência em caches globais.
- Eficiência nos custos, pois é possível utilizar os produtos e redes que já estão disponíveis a venda no mercado.

Como desvantagens:

- O programador se torna responsável por detalhes associados à comunicação de dados entre processos. Pode ser difícil de se mapear estruturas baseadas em memória global, usando este arranjo.

- Temos também o tempo de acesso não uniforme a dados, por exemplo, dados em nós distantes tem um tempo de acesso muito grande em relação aos nós locais.

## 2.2 Sistemas Operacionais De Tempo Real

Segundo (SILBERSCHATZ et al., 2013) o sistema operacional é um programa que gerencia o hardware de um computador. É também responsável por providenciar a base para os programas de computador além de agir como um intermediário entre o usuário do computador e o hardware do computador. Cada sistema operacional pode ser desenhado para atender um determinado objetivo, por exemplo sistema operacional em mainframes é feito para tirar o máximo de proveito da utilização do hardware. Já os PC, computadores pessoais, para jogos complexos, aplicações comerciais e outras atividades gerais.

Para os sistemas de tempo real (STR), tem-se outra variável importante a se considerar: o tempo decorrido a partir do momento que o sistema recebe o estímulo, até o tempo em que a saída é gerada. Portanto, além de se gerenciar os recursos de hardware e intermediar a interação com o usuário, o sistema deve escalonar as tarefas críticas de forma a gerar saídas respeitando a tempos pré-determinados. Geralmente sistemas de tempo real mantêm uma interação contínua com o ambiente. Conforme Figura 2.4:

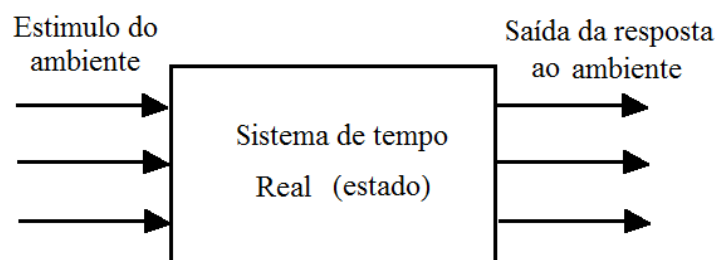


Figura 2.4 - O STR reage a entrada do ambiente produzindo uma saída que afeta o ambiente [Adaptada de (LABROSSE et al., 2008)]

Além de gerar saídas corretas um STR deve se ater às restrições de tempo ou sofrer consequências significativas para o sistema. Quando essas consequências consistem em degradação do desempenho sistema, mas não em uma falha, o sistema é classificado como sistema de tempo real “*soft*”. Já no caso de as consequências serem uma falha o sistema é classificado como sistema de tempo real “*hard*” (por exemplo o freio ABS em um automóvel).

Uma função de sistema (hardware, software ou combinação de ambos) é considerada uma função de tempo real “*hard*”, se e somente se o tempo limite precisa ser sempre atendido, caso contrário a tarefa falha. Um sistema pode ter uma ou mais tarefas de tempo real *hard*, assim como outras tarefas que não são de tempo real. Isso é aceitável, desde que o sistema agende propriamente as tarefas atendendo sempre o tempo limite das funções de tempo real “*hard*” (LABROSSE et al., 2008).

### **2.2.1 Diferenças entre Sistema Operacional de Tempo Real e Sistema de Tempo compartilhado**

Sistemas de tempo real diferem de sistemas de tempo compartilhado em três áreas fundamentais:

- Alto grau de escalonabilidade – requerimentos de tempo do sistema precisam ser satisfeitos a elevados graus de uso de recursos.
- Pior caso de latência – assegurar-se que o sistema ainda irá funcionar mesmo no pior caso de resposta a eventos.
- Estabilidade sob sobrecarga transitória – Quando o sistema é sobrecarregado por eventos e se torna impossível atingir todos os tempos limites, o tempo limite das tarefas críticas precisa ser garantido.

Na Tabela 2.1 temos a comparação de sistemas de tempo compartilhado *versus* sistemas de tempo real.



**Tabela 2.1 - Comparação entre Sistemas de Tempo Compartilhado e de Tempo Real.**  
[Adaptado de: (LABROSSE et al., 2008)].

Característica	Sistema de Tempo Compartilhado	Sistemas de Tempo Real
Capacidade do sistema	Alta vazão	Escalonabilidade e capacidade das tarefas do sistema cumprirem seu tempo limite
Responsividade	tempo médio de resposta rápido	pior caso de latencia assegurado, que é o pior caso de tempo de resposta a eventos
Sobrecarga	divisão justa	Estabilidade - quando o sistema é sobrecarregado, tarefas importantes tem seu prazo contemplado enquanto outras são adiadas

## 2.3 Arquiteturas Multicores

### 2.3.1 Motivação para Desenvolvimento

Com a aperfeiçoamento da tecnologia de fabricação de processadores, aproveitando-se da tendência predita pela lei de Moore, os fabricantes passaram a agregar mais de uma unidade de processamento a um único chip. Como essa unidade de processamento representa a estrutura de um processador típico passou a ser denominada de *core* ou núcleo.

Com o advento dos processadores multicores, entre 2005 e 2006 novas questões surgem nas comunidades de desenvolvedores (LIN; SNYDER, 2009):

- Durante muito tempo a melhoria de desempenho baseava-se exclusivamente na melhoria do hardware fazendo com que as técnicas de desenvolvimento de software paralelo pouco evoluíssem;
- Os programas existentes até então não podiam tirar proveito dos multicores diretamente;
- Programas que não exploram os chips multi-core, não experimentam nenhuma melhoria no desempenho;
- Muitos programadores da época não sabiam como escrever programas paralelos.

Isso despertou a necessidade de mudança da comunidade e busca por desenvolvimento de novas técnicas, bibliotecas e suporte em linguagens para a escrita de programas paralelos e reescrita de programas existentes.

### 2.3.2 Hardware

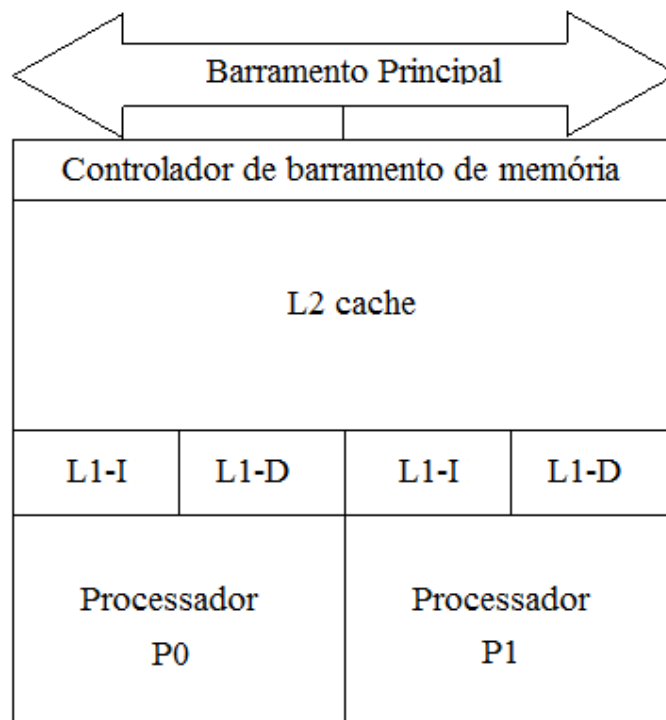
A IBM em 2002 foi a primeira fabricante a anunciar um processador multicore com o PowerPC 970, em maio de 2005 a AMD introduz o Dual Core Opteron e a Intel apresenta o Core Duo Pentium em janeiro de 2006.

#### 2.3.2.1 Intel Core Duo

As seguintes características são encontradas no processador Intel Core Duo:

- 2 processadores de 32 bits em um chip;
- Cada processador tem seu próprio 32K L1 cache de instrução e memória;
- 2MB ou 4MB de L2;
- Comunicação rápida entre dois processadores através de memória compartilhada;
- Controlador de memória, controlador entrada e saída.
- Comunicação rápida entre processadores através de memória compartilhada.

O projeto do Intel Core Duo é baseado na arquitetura do Pentium M, portanto um programa com um thread irá executar com velocidade comparável ao processador Pentium M. Em termos de codificação, existe consistência de dados entre os núcleos quando uma posição da memória é acessada. Isso ocorre porque quando o controlador de barramento faz uma requisição na memória, o dado é copiado para o cache L2, cache este compartilhado em ambos os núcleos. Após isso, ocorre a transferência para o cache L1 do processador que trabalhará com a instrução. Na *Figura 2.5* temos a estrutura lógica do processador Intel Core Duo o lado frontal do barramento faz a conexão com a memória RAM.



**Figura 2.5 - Organização lógica do Intel Core Duo.**  
[Adaptado de:(LIN; SNYDER, 2009)]

Uma possível complicação ocorreria quando os dois núcleos tentam alterar o valor numa mesma posição de memória, gerando inconsistências nos dados. Nesse caso, entra em ação o protocolo MESI (*Modified, Exclusive, Shared and Invalid* – Modificado, Exclusivo, Compartilhado e Invalido) que através desses quatro estados gerencia o conflito entre os threads. Embora o protocolo resolva os problemas de coerência permitindo com que o uso dos threads ocorre convenientemente, é acrescentado um atraso por conta da verificação do protocolo e a duplicação do consumo da banda do acesso à memória. Para contornar essa situação a Intel dobrou a média de banda disponível no Core Duo (LIN; SNYDER, 2009).

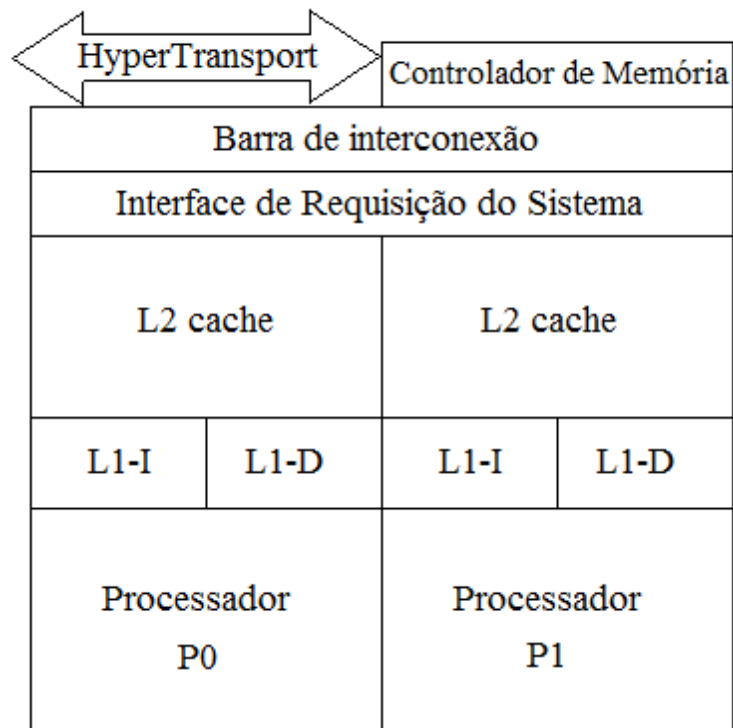
### 2.3.2.2 AMD Dual Core Opteron

As seguintes características são encontradas no processador AMD Dual Core Opteron:

- Dois processadores AMD 64 bits num único chip;
- Cada processador possui 64k L1 de cache para dados e instruções;

- 1MB de L2 Cache separados para cada núcleo;
- Arquitetura de conexão direta para acesso de memória compartilhada;
- Comunicação rápida no próprio chip entre os dois núcleos através da Interface de Requisição do Sistema.

O Sistema de Interface de Requisição do Sistema (SRI) é responsável pela coerência da memória, fazendo com que o sistema veja uma única imagem. O protocolo usado é MOESI, que assemelha ao MESI adicionando o estado *owned* (pertencido). Esse estado permite os valores de cache sejam compartilhados entre processadores, mesmo quando os dados da memória RAM estão desatualizados. Os acessos a memória RAM são realizados através do padrão da indústria HyperTransport technology. Na *Figura 2.6* temos a representação do processador AMD.



**Figura 2.6 - Organização lógica do AMD Dual Core Opteron.[Adaptado de: (LIN; SNYDER, 2009)]**

### 2.3.2.3 Comparação entre Intel e AMD

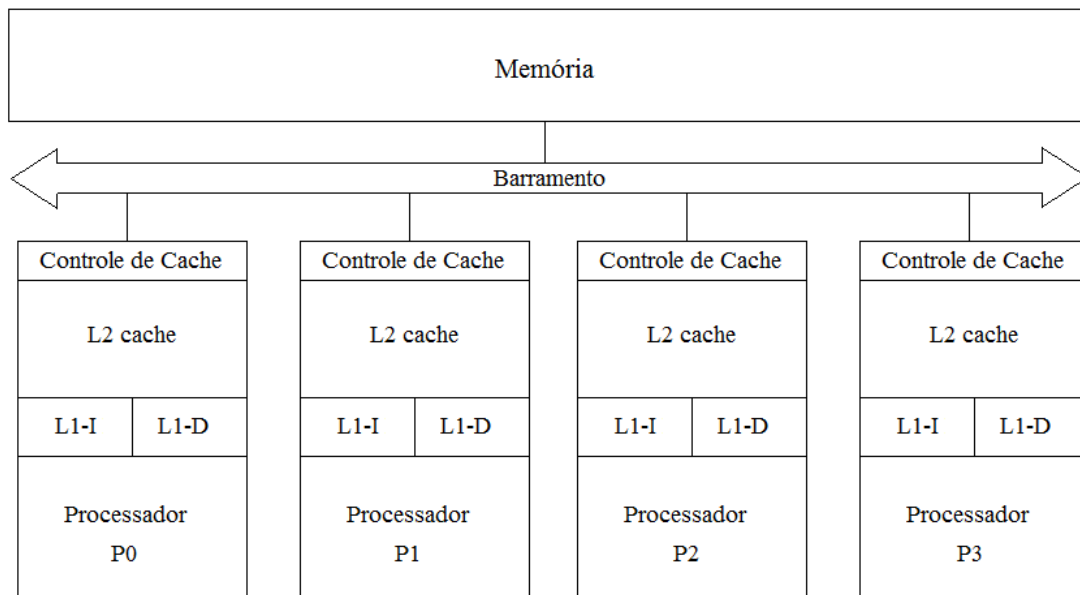
A principal diferença entre os processadores está no cache L2. O processador AMD é um processador privado, enquanto o Intel é compartilhado. O fato da coerência SRI atrás de L2, permite maior memória privada e facilidade de que a coerência seja combinada entre outros processadores (essa característica é da arquitetura conhecida como multiprocessador simétrico).

A característica do processador Intel de ser compartilhado, faz com que tenha baixa latência na comunicação interna do chip. Por isso, nessas arquiteturas apresentadas, um sistema que tenha apenas um processador (de dois núcleos) será mais conveniente a arquitetura da Intel, já um sistema que tenha mais de um processador (de dois núcleos cada) a arquitetura AMD será a mais adequada. Na perspectiva do programador essas diferenças são praticamente imperceptíveis, sendo vistas apenas como implementação em memória compartilhada.

### 2.3.2.4 Arquiteturas de Multiprocessadores Simétricos

Segundo (LIN; SNYDER, 2009) os chamados Multiprocessadores Simétricos (SMP) são computadores paralelos que acessam uma única memória lógica sendo esta porção de memória localizada fisicamente próxima ao processador. Para se ter uma visão consistente da memória, existe um ponto de conexão comum a todos processadores: o barramento de memória. Nesse barramento, cada processador pode verificar a atividade na posição de memória abaixo na *Figura 2.7* tem-se a representação lógica do SMP.

Os controladores de cache sempre verificam através do barramento de memória, se o endereço está referenciado por outros processadores e ajustando o marcador de controle nos seus valores de cache, para assegurar a coerência do uso do cache.



**Figura 2.7 - Diagrama lógico de Multiprocessadores Simétricos. [Adaptada de: (LIN; SNYDER, 2009)]**

O processador P1 pode fazer, por exemplo, a requisição ao bloco y na memória e supondo que o processador P2 já tenha esse bloco em cache, quando o P2 verifica o barramento fica ciente da requisição e assinala o marcador de controle como “compartilhado”. Se o processador P3 faz a requisição para escrever no bloco y, então tanto o P1 e P2 veem essa requisição e invalidam sua própria cópia, deixando que apenas o P3 tenha acesso. No momento em que P3 finalizar a escrita, a memória será atualizada e refletirá as requisições ao bloco y com o novo valor.

O procedimento de verificação no barramento garante a coerência, porém se torna um ponto de limitação uma vez que todos os processadores concorrem para utilização desse meio. Por essa razão computadores SMP geralmente tem menos que 20 conexões no barramento de memória (LIN; SNYDER, 2009).

As máquinas SMP atingem seu melhor desempenho em duas situações:

- Quando são pequenas e agrupadas próximas do barramento de memória;

- Quando tem um protocolo de cache eficiente, reduzindo assim as requisições ao barramento diminuindo o congestionamento do mesmo.

### 2.3.2.5 Arquitetura de chips heterogêneos

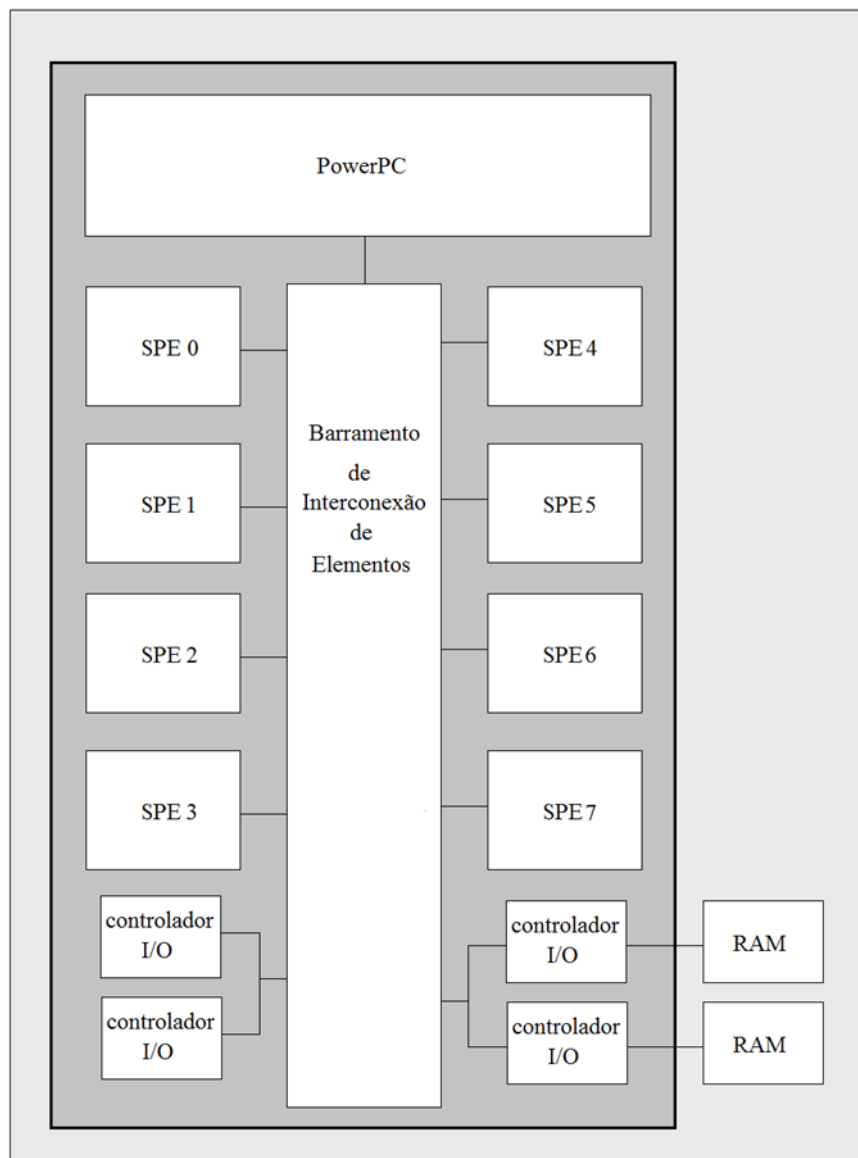
Diferente da arquitetura SMP que replicava processadores idênticos, na arquitetura heterogênea existe a possibilidade de agregar processadores adjuntos (*attached processors*) que podem executar tarefas mais especializadas. Nessa estrutura o processador padrão executa as instruções gerais difíceis de se paralelizar enquanto o processador adjunto executa a parte intensiva computacional. Alguns exemplos que seguem esse padrão:

- Unidades de Processamento Gráfico (GPU);
- FPGAs (*Field Programmable Gate Array*- Arranjo de Portas Programável em Campo);
- Processador Cell - usado em vídeo games.

O processador Cell desenvolvido em conjunto pela Sony, IBM e Toshiba. Tem a seguinte estrutura:

- Processador PowerPC 64 bits e *dual-threaded*;
- Oito SPE (*Synergistic Processing Elements* – Elementos de processamento sinérgico) capaz de executar instruções vetoriais;
- Cada SPE tem 256KB no chip de RAM;
- Alta velocidade no Barramento de Interconexão de Elementos conectando os SPEs.

O processador Cell não providencia a coerência de memória para os SPEs, nessa arquitetura foi escolhida simplicidade e desempenho ao invés de conveniência ao se programar. Na *Figura 2.8* temos o esquema do processador Cell.



**Figura 2.8 - Arquitetura processador Cell**

O processador PowerPC é o núcleo da linha de múltipla execução sendo responsável também pelo controle dos SPEs. O SPE é um processador RISC com organização SIMD de 128 bits (KAHLE et al., 2005).

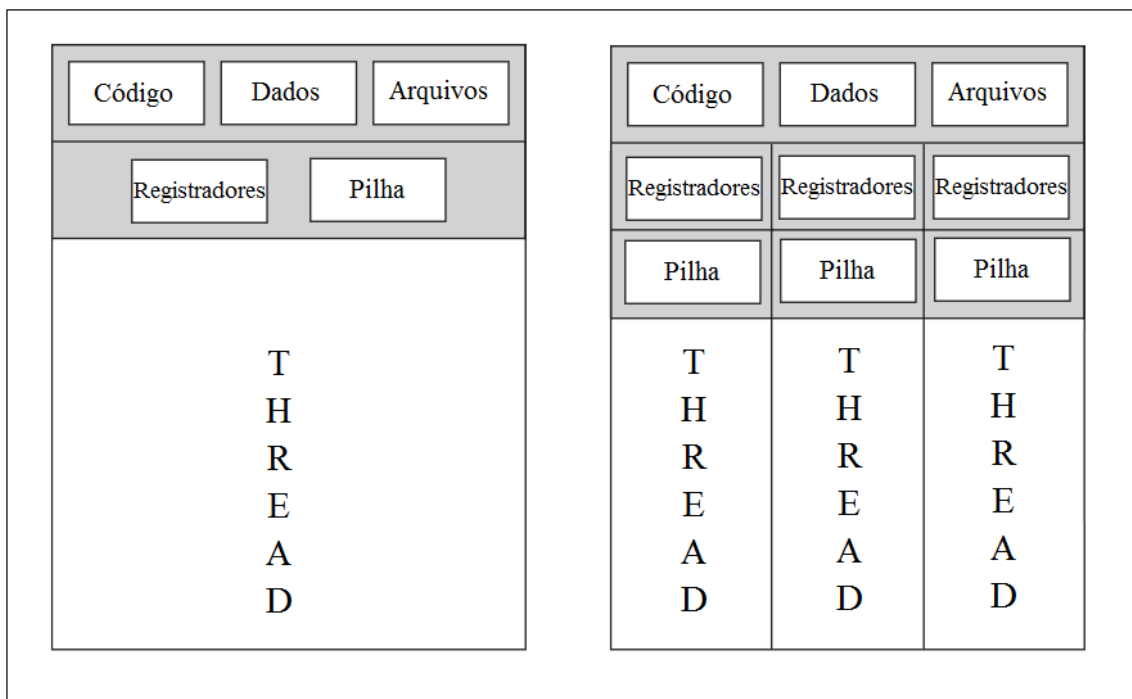
A complicação nessa arquitetura é gerenciar corretamente os dados até os SPEs e manter todos funcionando, quando atingido esse objetivo os processadores Cell produzem vazões surpreendentes.



### 2.3.3 Modelos de Programação Paralela

#### 2.3.3.1 Modelo de Threads

O thread é a menor unidade de utilização de um processador, cada thread é formado por número de identificação ID de thread, contador de programa e sua própria pilha. Ela também compartilha com outros threads recursos do processo a que pertence como: seção de código, dados, recursos do sistema operacional (SILBERSCHATZ et al., 2013). Na *Figura 2.9* é apresentado o processo com um thread e processo *multithreaded*.



**Figura 2.9 - À esquerda um processo com um thread e a direita um processo *multithreaded*.**

A utilização de threads beneficia o sistema computacional com:

- **Responsividade:** as tarefas do processo são divididas em sub-rotinas e cada sub-rotina é executada em um thread diferente. Com isso, caso um thread demore para responder ou esteja bloqueado, as demais sub-rotinas continuam funcionando normalmente, pois os threads trabalham independentemente entre si.
- **Partilhamento de recursos:** os threads compartilham diretamente recursos do processo;

- **Economia:** é mais custoso criar processos do que novas threads, isso se deve ao compartilhamento de recursos. Como consequência da diferença de tamanho, a troca de contexto nos próprios threads é muito mais rápida também;
- **Escalabilidade:** em um sistema *multithread* é possível alocar um thread para cada processador disponível, tirando grande proveito da arquitetura *multicore*.

O esforço dos fabricantes para uma padronização gerou duas principais implementações: POSIX Treads e OpenMP.

### **2.3.3.1.1 POSIX Treads**

POSIX thread (*Portable Operating System Interface thread*), conhecido também como Pthreads, surgiu como padrão em 1995 especificado pela *IEEE POSIX 1003.1c standard*.

Os Pthreads são implementados na linguagem C através do arquivo *header* ou biblioteca (Pthread.h). Essa biblioteca oferece suporte para criação e destruição de threads, além de algumas outras funções de coordenação de threads como: *locks*, *mutex*, seções críticas, semáforos e variáveis condicionais.

Nesse modelo de programação a memória *heap* é alocada dinamicamente e compartilhada entre os demais threads. Os programadores devem gerenciar corretamente o uso de dados compartilhado de forma a evitar *deadlocks* e condições de corridas (CHOUGULE; GUTTE, 2014).

### **2.3.3.1.2 OpenMP**

OpenMP (*Open Multi-Processing*) é padronizada pelo consórcio OpenMP Architecture Review Board dos quais participam as empresas como AMD, IBM, Intel, Cray, Fujitsu, Nvidia, Red Hat, Texas Instruments, Oracle e outras.

OpenMP é uma API que suporta a plataforma de memória compartilhada, multiprocessamento nas linguagens C, C++ e Fortran. Diferente de Pthreads que é estruturada através de bibliotecas, OpenMP fornece um conjunto de diretivas de compilação, “*pragmas*” que orientam o gerenciamento das threads e biblioteca de rotinas. As principais vantagens são a escalabilidade e alta portabilidade, podendo suportar desde computadores pessoais até a supercomputadores.

Nessa padronização o gerenciamento dos threads é feito de maneira implícita. Um thread “mestre” é responsável por dividir as tarefas para um número específico de threads “escravos”, esses threads rodam de forma concorrente podendo ser alocados em diferentes processadores. Terminado as tarefas os threads são juntados novamente em um só (thread mestre). É possível determinar quantos processadores, número de threads, mapeamento de seções críticas, sincronizações entre outras, através de diretivas de compilação ou parâmetros, ou funções. (T.C et al., 2011).

### 2.3.3.2 Modelo de Passagem de mensagens

Esse modelo de programação paralela consiste na troca de mensagens entre tarefas que podem estar na mesma máquina, ou em diferentes máquinas, sendo utilizado principalmente para arquitetura de memória distribuída.

As funções `envia()` e `recebe()` controlam o fluxo de comunicação, a passagem de mensagem é bidirecional e as tarefas envolvidos precisam colaborar para a transferência de dados. Na *Figura 2.10* o esquema das atividades de comunicação é esquematizado.

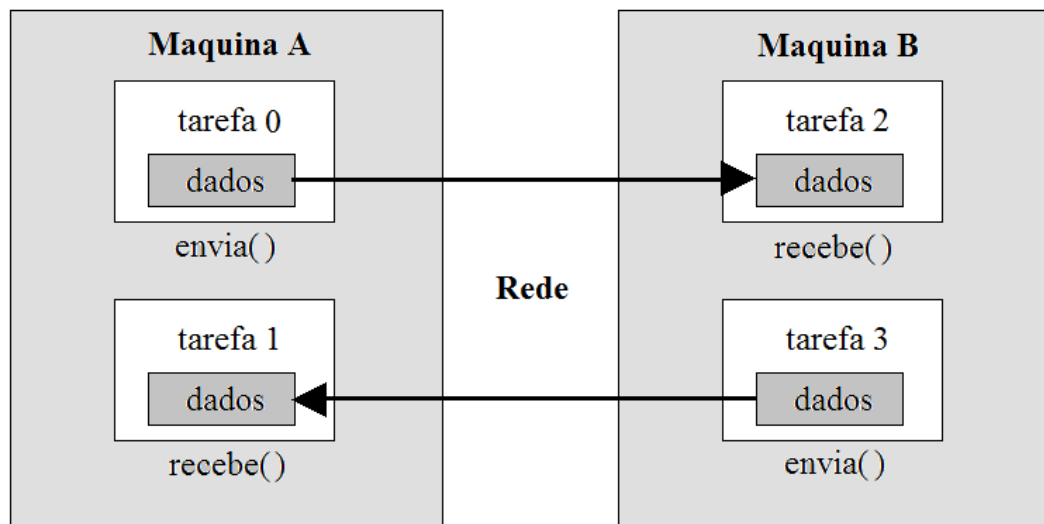


Figura 2.10 - Esquema do modelo de passagem de mensagens

Com relação à estrutura das memórias existem dois tipos de abordagem e são usadas duas formas de movimentação: referências a memória no caso do uso de memórias locais, e passagem de mensagens no caso de acesso não local de memória. Devido as chamadas padronizadas e bem definidas, esse padrão é

considerado mais fácil de se depurar do que os padrões de memória compartilhada (LIN; SNYDER, 2009).

### **2.3.3.2.1 Interface de Passagem de mensagens (MPI)**

O MPI (*Message Passing Interface*) é um padrão de bibliotecas baseado no consenso de fabricantes, pesquisadores, desenvolvedores de software e usuários. Embora, não seja um IEEE ou ISSO, é considerado um padrão da indústria para escrita de aplicações HPC (*High Performance Computing*).

MPI fornece portabilidade, eficiência, padronização e funcionalidades, sendo possível passar mensagens ponto a ponto, como também mensagens globais. Além disso, fornece padrão para bibliotecas de escrita, depuração e teste de desempenho. As implementações estão disponíveis nas linguagens C, C++ e Fortran e suas distribuições mais conhecidas são: GridMPI, LAM/MPI, OpenMPI, MPICH e MVAPICH.

A divisão das tarefas ocorre de maneira análoga ao modelo Pthreads. Apesar, do gerenciamento implícito de threads, cabe ao programador mapear quais tarefas serão executadas e por quais processos. A comunicação usa o modelo de trocas de mensagens entre processos descritos no item 2.3.3.2 de passagem de mensagens.

A vantagem para o usuário é que MPI é padronizado em vários níveis, por isso pode-se utilizar a mesma sintaxe não importando a implementação. Cada chamada MPI deve comportar-se de maneira semelhante independente da implementação, garantindo a portabilidade de aplicações paralelas. (YANG et al., 2009).

Devido a sua característica geral, MPI favorece a sua utilização em sistemas distribuídos, apesar de ser possível também utilizá-lo em sistemas de memória compartilhada (porém com degradação no desempenho em relação ao modelo de memória compartilhado).

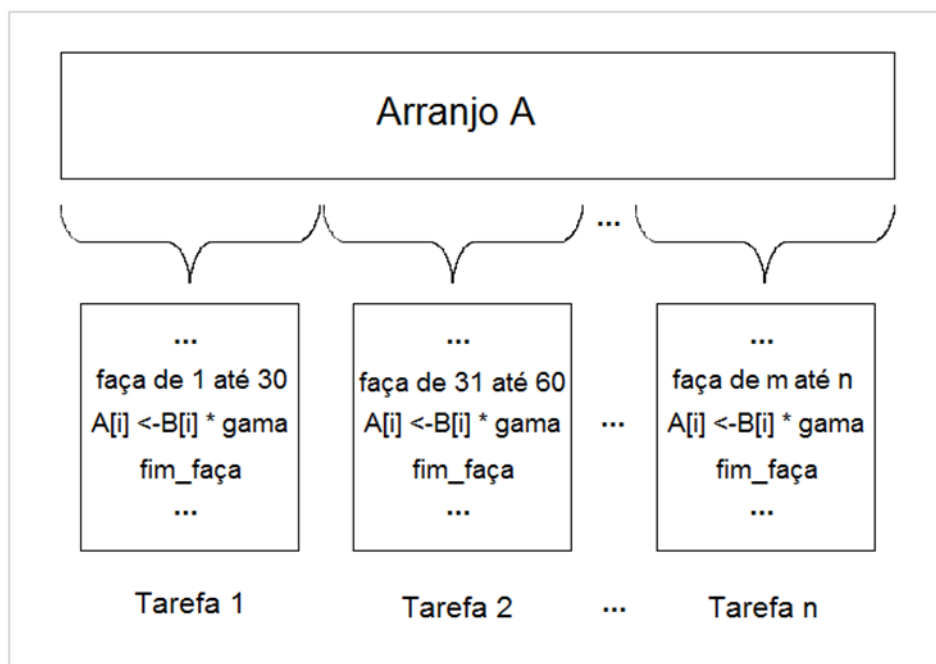
### **2.3.3.3 Modelo PGAS**

O PGAS (*Partitioned Global Address Space*- Espaço de endereço Global Particionado) é um modelo de programação paralelo que visa melhorar a

produtividade na programação e aumentar o desempenho da aplicação. A principal ideia desse modelo é que o compartilhamento de endereço global favorece a produtividade, porém é necessário que se faça diferenciação entre acessos a dados locais e remotos de forma a se realizar otimizações de desempenho e suporte a escalabilidade em arquiteturas paralelas de larga escala (WAEL et al., 2015).

Como principais características, esse modelo de dados paralelo possui (Barney, 2015):

- Espaço de endereço global;
- A maior parte do trabalho paralelo é feita buscando desempenho em operações de conjunto de dados, como arranjos ou cubo;
- Um conjunto de tarefas trabalha coletivamente em uma mesma estrutura de dados, entretanto cada tarefa trabalha em uma porção diferente dessa mesma estrutura;
- Tarefas executam a mesma operação em sua partição. Na *Figura 2.11* temos um esquema que representa o modelo PGA.



**Figura 2.11 - Esquema de operação PGAS – [Adaptado de: (BARNEY, 2015)]**

Na década de 1990 surgem as implementações de PGA: Co-Array como extensão de Fortran, Titanium extensão de Java e Unified Parallel C para a linguagem C.

Por volta da década de 2000 surgem três novas linguagens: Chapel, X10 e Fortress. Essas linguagens aparecem devido ao projeto HPCS (High Productivity Computing Systems gerenciado pelo DARPA (*Defense Advanced Research Projects Agency*- dos Estados Unidos), tendo como foco agregar em relação as linguagens anteriores: o aumento na produtividade, facilidade de programação, alto desempenho, portabilidade.

Entretanto, apesar do entusiasmo promovido pela criação dessas linguagens elas acabaram não se difundindo. Isso se deu ao seu nicho específico de aplicações (por exemplo, as aplicações que usam arranjos globais), ou de não provarem um maior impacto em sua utilização.

## 2.4 Mecanismos de comunicação entre processos

Um programa de computador enquanto permanece em seu dispositivo de armazenamento é uma entidade passiva. No entanto, quando é carregado para a memória passa a ser um elemento ativo de execução tornando-se um processo.

Os processos possuem um contador de instruções que aponta para próxima instrução a ser executada. Têm sua própria pilha que contém informações temporárias, sobre variáveis e parâmetros. A seção de dados contém as variáveis globais, além disso os processos têm uma porção do *heap* que é alocada em tempo de execução.

Um processo pode ser classificado como independente ou colaborativo. No processo independente nenhum outro processo afeta ou é afetado pelo mesmo. Já no processo colaborativo o mesmo pode afetar ou ser afetado por outro processo executado no sistema (SILBERSCHATZ et al., 2013).

O termo IPC (*Interprocess Communication*), refere-se aos mecanismos utilizados para comunicação entre processos. As duas principais estratégias de

comunicação entre processos são: o compartilhamento de memória e a troca de mensagens.

No paradigma de memória compartilhada, uma região de memória é explicitamente definida como compartilhada entre os processos. Essa região de memória compartilhada deve ser cuidadosamente gerenciada, a fim de se evitar inconsistência de dados. Além desse gerenciamento, operações do tipo “produtor-consumidor” devem ser sincronizadas corretamente de forma a se evitar o consumo de produtos inexistentes.

De uma forma geral, o modelo de memória compartilhada é mais rápido que o modelo de troca de mensagens, isso ocorre devido a uma necessidade menor de chamadas do sistema. Além disso, uma vez estabelecida a área compartilhada os próprios processos passam a gerenciar, não necessitando mais intervenção do *kernel* do sistema operacional (SILBERSCHATZ et al., 2013)

No modelo de troca de mensagens é possível se realizar comunicação entre processos mesmo no caso em que não compartilham endereços de memória. Esse tipo de comunicação se torna especialmente útil em sistemas distribuídos, uma vez que os processos não necessitam estar na mesma máquina para se comunicar.

No paradigma de troca de mensagens é possível que se envie mensagens de tamanho fixo ou variável. Uma ligação de comunicação deve existir entre os processos, que pode ser memória compartilhada, barramento de hardware ou rede. A comunicação pode ser realizada diretamente, nesse caso deve-se nomear o remetente ou/e destinatário. Para a comunicação indireta são usadas caixas de mensagens ou portas. A mensagem pode ser bloqueante ou não bloqueante, conhecida como síncrona ou assíncrona, respectivamente.

### 2.4.1 Comunicação cliente servidor

Para a estrutura cliente-servidor de IPC pode-se destacar como principais estratégias (SILBERSCHATZ et al., 2013)

- **Sockets:** nessa abordagem uma conexão é feita a concatenação de um endereço IP e um número de porta. A conexão é feita ponto-a-ponto usando protocolo TCP ou UDP através de uma rede.

- **RPC** (*Remote Procedure call*): é uma abstração de chamadas de procedimentos para sistemas conectados por rede. Nessa abordagem, um procedimento remoto é iniciado pelo cliente, enviando uma mensagem para um servidor remoto com o procedimento a ser executado. O comando é executado no servidor e a resposta é enviada ao cliente.
- **Pipe**: Um dos primeiros mecanismos de IPC a ser implementado em sistemas UNIX. Sendo um dos mecanismos mais simples de comunicação entre processos. Existem duas variantes, ambas sendo usadas para a comunicação em uma mesma máquina:
  - *Pipe* comum: o produtor escreve no fim do *pipe*, e o consumidor começa a ler a partir da outra extremidade. A comunicação é unidirecional, sendo necessário dois *pipes* para a comunicação nas duas direções. Sua existência está atrelada à comunicação entre dois processos. Sendo assim, tão logo termine a comunicação o *pipe* deixa de existir;
  - *Pipe* nomeado: nesse tipo de *pipe* a comunicação pode ocorrer de forma bidirecional, embora possa se utilizar uma direção de cada vez (*half-duplex*). Uma vez estabelecido o *pipe* nomeado pode ser utilizado na comunicação de diversos processos e continua existindo mesmo que a comunicação entre determinados processos termine.

## 2.5 Ferramentas para Programação paralela

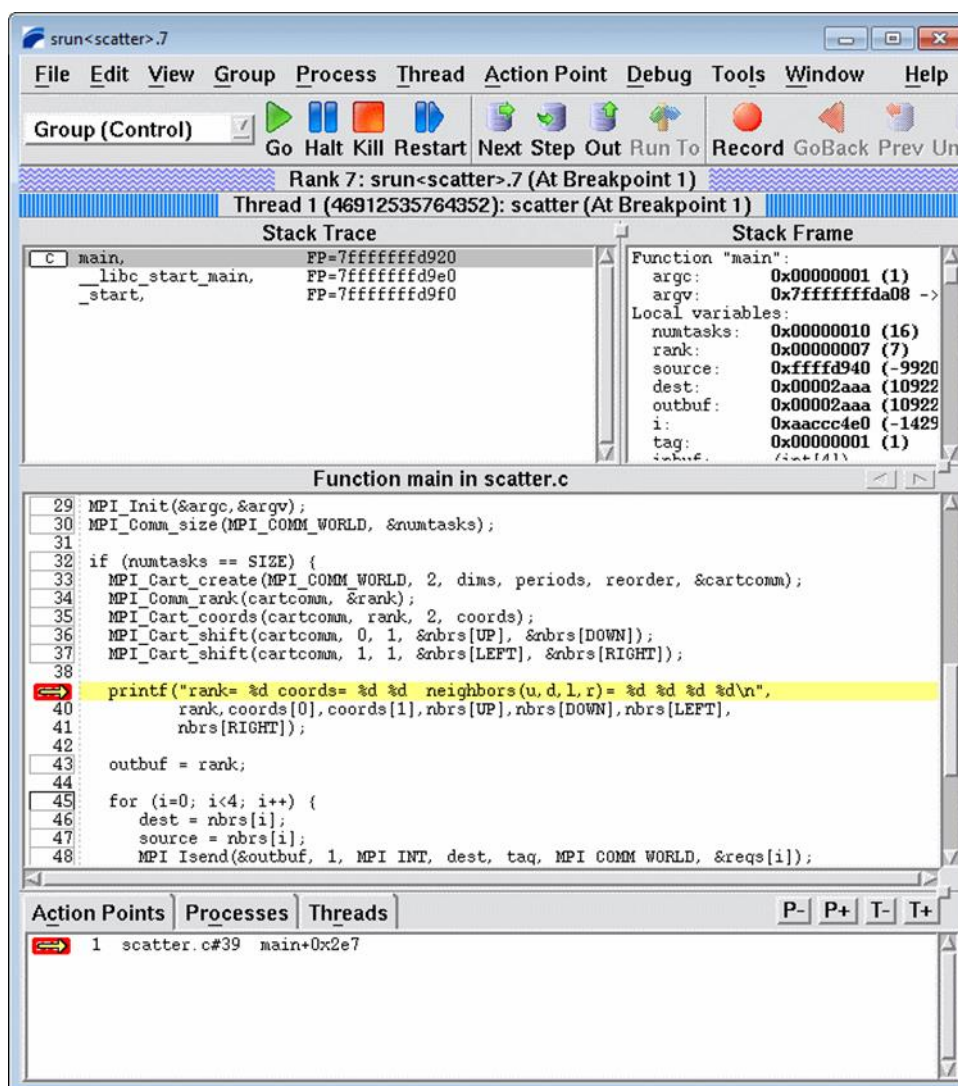
À medida que os processadores deixaram de buscar um aumento do *clock* como principal fator de desempenho e buscaram melhorar a sua própria estrutura interna, aumentando o número de núcleos, melhorando capacidade de instruções vetoriais, tornou-se importante o desenvolvimento de ferramentas que auxiliassem na produção de software que aproveitasse esse novo potencial.

Tarefas que antes eram simples de serem implementadas em código sequencial se tornaram mais complexas no contexto paralelo. Necessidades adicionais surgiram como sincronizações, controles de regiões críticas, vazamento de memória, entre outros. Entre as ferramentas que se destacaram por possuir interface visual, diversos recursos para gerenciar características do contexto paralelo e facilidade de uso podemos citar: TotalView, Alinea DDT e Intel Parallel Studio XE.

TotalView, mantida pela Rogue Wave Software, Inc, é uma ferramenta sofisticada para inspeção e análise de código (inclusive multithread). A TotalView fornece inspeção para códigos seriais, paralelos, multi-processos, *multithreads*,



aplicações aceleradas por GPU e aplicações híbridas escritas em C/C++ e Fortran, sendo possível utilizar tanto linha de comandos, quanto interface visual. Pode-se inspecionar programas, processos rodando e realizar depurações na memória. Na *Figura 2.12* apresenta-se o ambiente TotalView e análise de código MPI.



**Figura 2.12 - Ferramenta TotalView analisando código MPI [Extraído de: (BARNEY,2015) ]**

Outra importante ferramenta é a Allinea DDT, produto da Allinea Software Ltd., para inspeção de código com interface gráfica fácil de se usar, sendo capaz de inspecionar aplicações como: *multithread*, OpenMP, MPI, softwares heterogêneos que utilizam GPU, paradigmas híbridos como MPI+OpenMP ou MPI + CUDA, possui recursos de análise sintática destacando os potenciais erros no código fonte, além de

vir com depurador de memória embutido, integração de filas de mensagens MPI. Seus recursos estão disponíveis para as linguagens C/C++ e os derivados de Fortran, modelos de PGAs como UPC (*Unified Parallel C*) e Fortran 2008 Co-arrays, linguagens de GPU como HMPP, OpenMP *Accelerators*, CUDA e CUDA Fortran. Alinea DDT pode ser usada desde desktops até supercomputadores. Na *Figura 2.13* temos a interface do Alinea DDT (“Alinea DDT”).

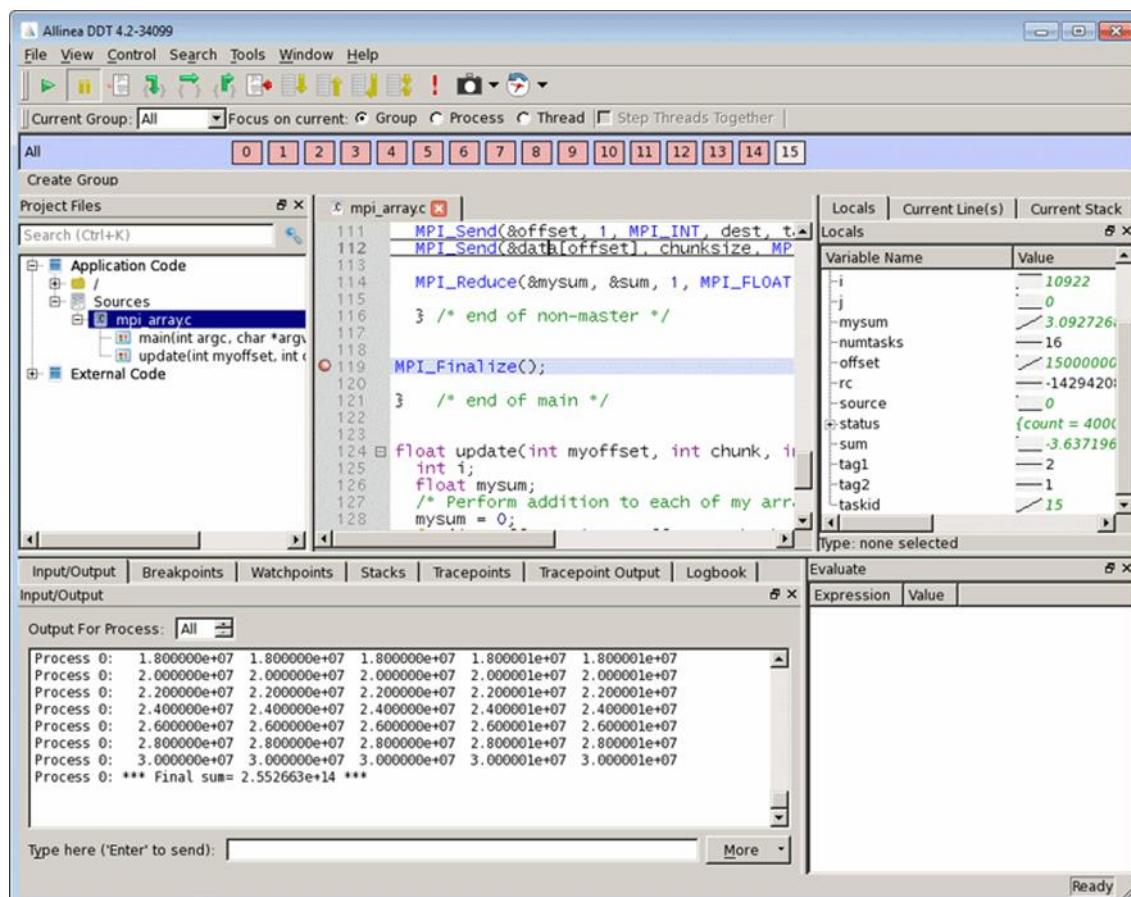


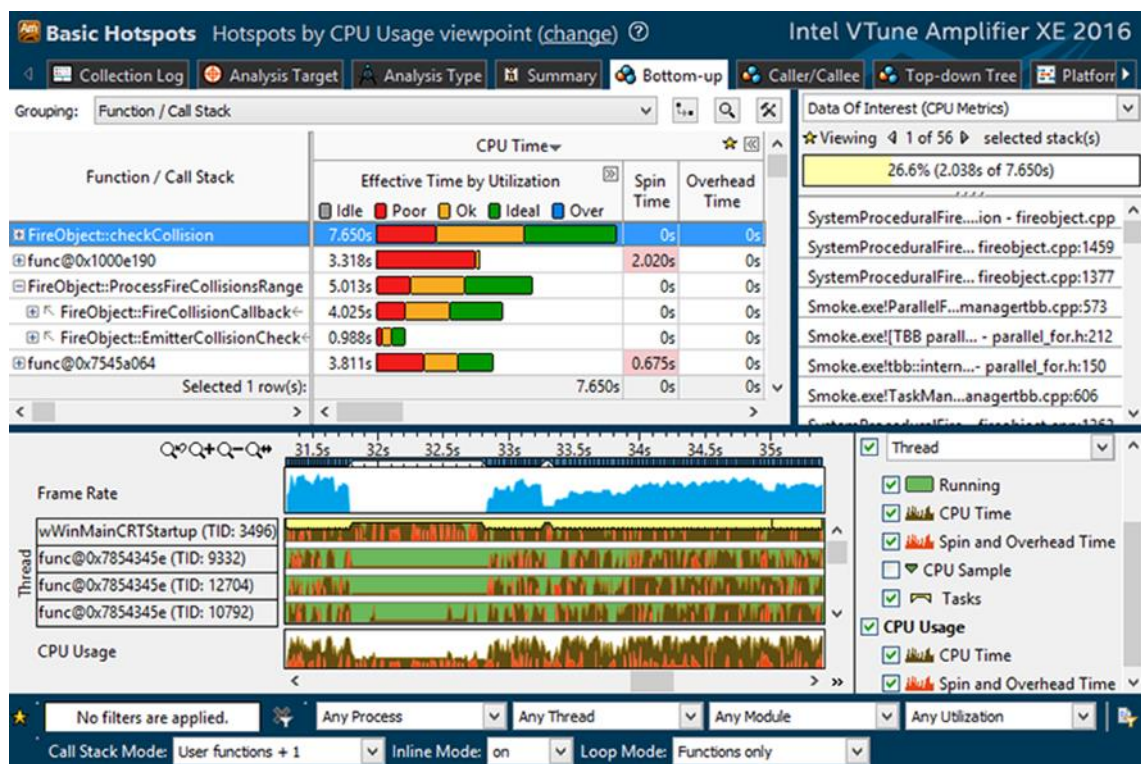
Figura 2.13 - Ferramenta Alinea DDT analisando código [Extraída de (“Alinea DDT”)]

Intel Parallel Studio XE é um produto desenvolvido pela Intel, para facilitar o desenvolvimento em Windows e Linux de aplicações C/C++ e Fortran para aplicações paralelas. O Intel Parallel Studio XE é constituído pelos seguintes componentes (BLAIR; STOKES, 2012):

- Intel Parallel Advisor: este componente dá conselhos de como adicionar paralelização ao programa.

- Intel Parallel Composer XE: o Composer XE contém o compilador e bibliotecas usadas para criar código otimizado e paralelo.
- Intel Parallel Inspector XE: é usado para checar os erros comuns de programa do tipo paralelo, tais erros como *deadlocks*, condições de corrida, erros de memória e vazamento de recursos tanto para programas paralelos como sequencias.
- Intel Vtune Amplifier XE – é utilizado para analisar uma determinada aplicação e identificar os pontos críticos e gargalos. Esse componente analisa também o quão eficiente um programa paralelo é.

Abaixo temos a *Figura 2.14* com do ambiente Intel Parallel Studio XE (“Intel® Parallel Studio XE 2016”).



**Figura 2.14 - Ferramenta Intel Parallel Studio XE [Extraída de (“Intel® Parallel Studio XE 2016”)]**

# Capítulo 3

## REVISÃO BIBLIOGRÁFICA

---

### 3.1 Arquiteturas multicore recentes

A partir do marco de 2002, quando a IBM lançou o seu primeiro computador multicore o PowerPC 970, e outras fabricantes entram com produto multicore no mercado; como AMD em 2005 (Dual Core Opteron) e Intel em 2006 (Core Duo Pentium), constantes melhorias foram adicionadas, mais núcleos agregados a um mesmo chip, instruções vetoriais ampliadas, melhorias em barramentos de comunicação e mecanismos de coerência de cache.

Abaixo são apresentadas melhorias presentes em duas arquiteturas de servidores recentes: Intel Xeon 2600 – Sandy Bridge (2012) e AMD Opteron 6200 – Bulldozer (2011) a partir de dados obtidos em (MOLKA et al., 2014).

- Intel Xeon 2600: é baseado na família da microarquitetura Sandy Bridge. Cada núcleo possui caches L1 e L2 dedicados, assim com sua própria FPU (*floating-point unit* - unidade de ponto flutuante). A FPU pode executar duas instruções de 256 bits por ciclo, uma adição e uma multiplicação. Através do HyperThreading dois threads podem ser executados por núcleo compartilhando a maioria dos recursos. Na *Figura 3.1* extraída de (MOLKA et al., 2014), observa-se oito núcleos (cores), 20MB de cache L3, um canal quádruplo e duas interconexões QPI (*QuickPath Interconnect*).

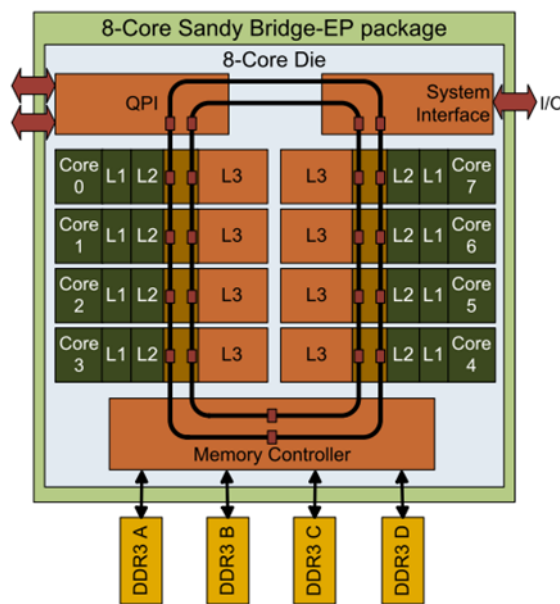


Figura 3.1 - Processador 8 núcleos Sandy Bridge-EP [Extraído de: (MOLKA et al., 2014)]

- AMD Opteron 6200:** é baseado na microarquitetura Bulldozer. Esta microarquitetura baseia-se em unidades de computação de dois núcleos, que compartilham entre si unidades de carregamento e decodificação de instruções, unidade de ponto flutuante, além de cache L1 de instruções e cache L2. Cada núcleo possui sua própria unidade execução de operação com inteiros e cache L1 de dados. Cada cache de dados L1 possui duas portas de leitura de 128 bits e uma porta de escrita de 128 bits. A FPU proporciona a fusão de instruções de multiplicação e adição e executa duas instruções de 128 bits por ciclo. As instruções AVX, de 256 bits são divididas em duas partes de 128 bits. Instruções SIMD também são executadas na FPU compartilhada. Na *Figura 3.2* temos a organização do AMD Opteron de dezesseis núcleos. Este processador consiste na ligação de dois blocos de oito núcleos através do barramento HyperTransport.

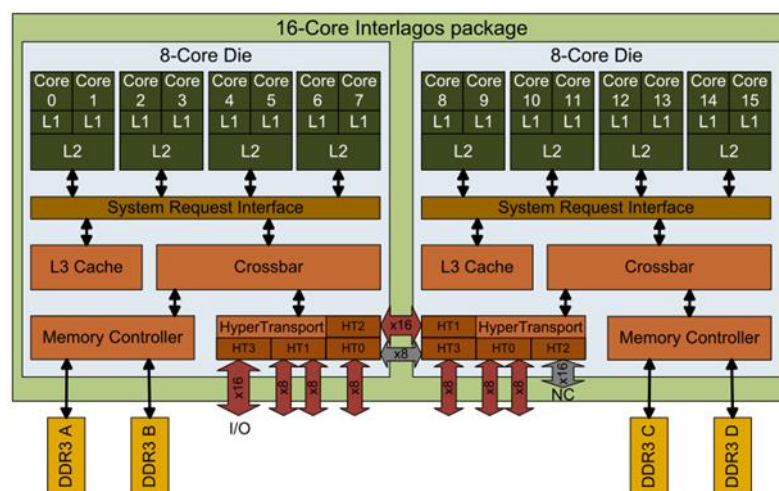


Figura 3.2 - Processador 16 núcleos Bulldozer. [Extraído de (MOLKA et al., 2014)].

## 3.2 Arquitetura multicore de sistema embarcado recente

Existem inúmeras arquiteturas disponíveis para sistemas embarcados, cada uma dependente do escopo de atuação desse sistema. Em sua maioria construídas em SOC (*System on Chip*) que agrega CPUs, GPUs, controladores, caches e outros componentes de acordo com a aplicação que foram desenhados.

Da mesma forma que a arquitetura de computadores, a arquitetura da CPU dos sistemas embarcados evoluiu para os processadores multicore. A família de processadores ARM (*Advanced RISC Machine*) tem sido o componente mais comum dos SOC e como exemplo de arquitetura recente temos o processador Cortex-A72.

O processador Cortex-A72 (“Cortex-A72 Processor - ARM”) é um processador de alto desempenho, lançado no início de 2015, baseado na arquitetura ARMv8, que tem de um a quatro *cores* (núcleos) SMP sendo capaz de executar instruções de 32 ou 64 bits. O processador tem disponível o recurso TrustZone que implementa em hardware o conceito da separação de aplicações seguras e não-seguras isoladas e os modos de operação, possui o recurso NEON para instruções SIMD avançadas de 128 bits e VFPv4 (*Vector Floating-Point*) que permite cálculos de precisão simples e dupla de ponto flutuante, além de suporte de hardware para virtualização, recurso CoreSight para inspeção e rastreamento em multicore. A *Figura 3.3* representa os módulos e características do Cortex– A72.

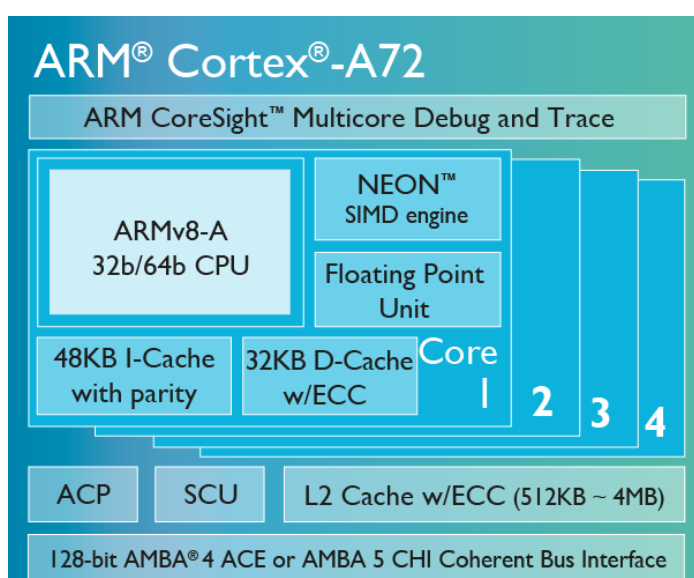


Figura 3.3 – Estrutura e componentes do Cortex – A72 [Extraído de: (“Cortex-A72 Processor - ARM”)]

O protocolo AMBA do processador Cortex-72 é um padrão aberto para a conexão e gerenciamento de blocos funcionais no SOC, além de facilitar o desenvolvimento de multiprocessador para um grande número de controladores e periféricos.

### 3.3 Métodos para programação paralela em multicores

Para se aproveitar o potencial disponibilizado pelos processadores multicore, novos paradigmas foram criados e conseqüentemente novas ferramentas. As principais abordagens visaram a criação de modelos voltados para memória compartilhada, passagem de mensagens e modelo de dados paralelos.

No modelo de memória compartilhada, o paralelismo ocorre através de posições de memória compartilhadas entre os diversos processadores através de um mesmo barramento local que conecta processadores a memória. São exemplos de ferramentas e bibliotecas de memória compartilhada:

- **Pthread**: é a biblioteca da linguagem C que define o uso de POSIX threads. Esta biblioteca inclui desde criação de threads e sua destruição, gerenciamento de regiões críticas, uso de *locks*, *mutex* e semáforos.
- **OpenMP**: é uma API de desenvolvimento para as linguagens C/C++ e Fortran, na qual o gerenciamento de threads ocorre de maneira implícita através de diretivas de compilação e uso de bibliotecas.

No modelo de passagem de mensagens, a organização, paralelização e divisão das tarefas ocorre por meio de trocas de mensagens. Esse modelo favorece a utilização de sistemas distribuídos, embora possa ser utilizado com arquiteturas de memória compartilhada. Como exemplo de padronização de passagem de mensagens:

- **MPI**: é um padrão de bibliotecas para passagem de mensagens, fornecendo suporte a mensagens ponto a ponto, como mensagens globais. Em MPI o gerenciamento é feito de maneira implícita, cabendo ao programador especificar quais tarefas serão executadas e em quais processos. As implementações desse padrão estão disponíveis para as linguagens C/C++ e Fortran. As distribuições mais conhecidas são: GridMPI, LAM/MPI, OpenMPI, MPICH e MVAPICH.

No modelo de dados paralelos, também conhecido como PGA (*Partitioned Global Address Space*), temos um espaço de endereço global. Os grupos de tarefas trabalham coletivamente executando uma mesma operação em diferentes áreas de um mesmo arranjo de dados. As principais implementações são: Co-Array Fortran, Titanium, Unified Parallel C, Chapel, X10 e Fortress.

A partir da combinação de modelos paralelos existentes descritos anteriormente, novas possibilidades vêm surgindo nos denominados modelos híbridos. A ideia é mesclar arquiteturas heterogêneas, buscando aproveitar melhor o poder computacional e recursos disponíveis:

- MPI + OpenMP: Nesse arranjo, os processadores com acesso à memória compartilhada utilizam OpenMP na comunicação entre núcleos, já na comunicação entre nós MPI é utilizado. Como vantagem possui melhor desempenho em relação ao MPI, assim como também uma economia de memória. Porém como desvantagem existe uma adição de complexidade de software e menor portabilidade (BULL et al., 2010).
- MPI + Pthreads: Neste arranjo é possível gerenciar com uma granularidade fina usando Pthreads, e uma granularidade grossa usando MPI. Esta abordagem permite um uso mais eficiente de multiprocessadores em sistemas distribuídos (PFEIFFER; STAMATAKIS, 2010).

Com o tempo, as GPUs (*Graphics Processing Unit*) tornaram-se capazes de além de renderizar gráficos, realizar processamentos gerais aproveitando assim sua estrutura de hardware altamente paralelizada para a computação paralela geral. Nas GPUs da fabricante nVIDIA, essa nova funcionalidade surge com a plataforma CUDA (*Compute Unified Device Architecture*). E de uma forma mais geral (não atrelado a fabricante), aparece o OpenCL (*Open Computing Language*) gerenciado pelo Khronos Group. No OpenCL tanto a CPU quanto a GPU trabalham juntas na execução de *kernels* permitindo o uso de paralelismo de tarefas e de dados.

Como exemplo de modelos híbridos usando GPU podemos citar:

- MPI+CUDA: Nessa estrutura as tarefas MPI rodam em processadores que usam a memória local e rede para comunicar-se entre si. Os *kernels* recebem as tarefas computacionalmente exaustivas rodando em nó da GPU, utilizando CUDA para a troca de dados entre memória e GPU (Barney, 2015).



### 3.4 Ferramentas para programação paralela em multicores de tempo real

Para a programação paralela em multicores de tempo real, além de características como gerenciamento de recursos compartilhados, seções críticas, comunicação e sincronização, restrições de tempo de resposta passam a ser essenciais para o funcionamento correto do dispositivo.

No sistema operacional Linux a latência do *kernel* é a soma das latências: de interrupção, duração de manipulador (*handler*), latência do agendador (*scheduler*) e o período de atuação do agendador. O Linux é considerado um sistema operacional preemptivo, no entanto, quando uma interrupção ocorre enquanto uma tarefa executa uma chamada de sistema, a chamada de sistema precisa ser finalizada antes que outra tarefa possa ser agendada. Por isso, por padrão o *kernel* do Linux não realiza preempção de *kernel*.

Ingo Molnar, Thomas Gleixner e Steven Rosted lideraram o projeto PREEMPT\_RT, que foi a primeira abordagem a tornar o Linux tempo real. A partir da versão 2.6, novas características foram adicionadas como preempção de *kernel* e melhor suporte a APIs de tempo real POSIX. Como principais modos de preempção temos (“Linux kernel configuration for 2.6.18 on i386”):

- CONFIG\_PREEMPT\_NONE: neste modo código do *kernel* nunca sofre preempção. Sendo melhor para sistemas com cálculos intensivos, em que alta vazão é a característica desejada.
- CONFIG\_PREEMPT\_VOLUNTARY: o *kernel* voluntariamente pode realizar uma preempção em si mesmo. Abordagem vantajosa para computadores pessoais que requerem uma reação rápida para as entradas do usuário, leve diminuição na vazão (*throughput*) do sistema.
- CONFIG\_PREEMPT: boa parte do código do *kernel* pode sofrer preempção a qualquer tempo (exceção a áreas críticas do *kernel*). Pode ser usado para computador pessoal ou sistemas embarcados com requisitos de atraso na faixa de milissegundos.

Como vantagens da abordagem Linux PREEMPT\_RT, podemos citar um atraso mínimo, tornando o sistema mais determinístico. Isso porque interrupções de tempo real não podem ser bloqueadas pelo Linux e sim apenas por um subsistema de tempo real. Como desvantagens temos: as aplicações de tempo real rodam no espaço de

*kernel* (caso haja um problema, o sistema todo falha), comunicação entre subsistema de tempo real e Linux não podem ocorrer em tempo real (LIPARI, 2008).

Outra ferramenta de desenvolvimento em sistemas em tempo real é Xenomai, em que diversas APIs de sistemas de tempo real são disponibilizadas para plataformas Linux. Os principais objetivos de Xenomai são auxiliar (“Start Here – Xenomai”):

- Projeção, desenvolvimento e funcionamento de aplicações de tempo real no Linux;
- Migração de uma aplicação proprietária de RTOS para Linux;
- Execução de aplicações RTOS (VxWorks, pSOS, VRTX, uITRON, POSIX) de forma otimizada ao lado de aplicações Linux.

De forma a atender os requisitos de uma aplicação de tempo real Xenomai usa duas opções:

- Núcleo Cobalt: é adicionado ao *kernel* do Linux como *co-kernel*, lidando com todas as atividades de tempo crítico, como manipulador de interrupções, e agendador de threads em tempo real. O núcleo Cobalt tem maior prioridade sobre as atividades do *kernel* nativas. Nesta configuração dual de *kernel*, todas APIs RTOS Xenomai fornecem uma interface com o núcleo Cobalt, além das API consideradas de tempo real, incluindo o subconjunto de serviços POSIX 1003.1c implementados por Xenomai (como libcobalt). Representado na *Figura 3.4*:

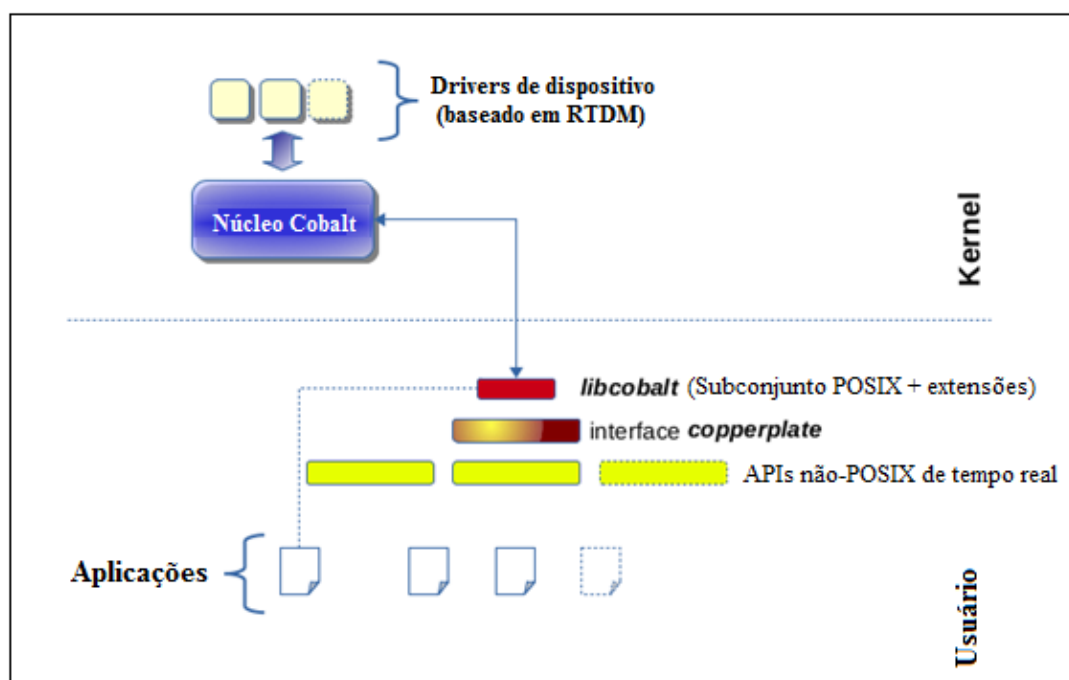


Figura 3.4 - Configuração Xenomai dual de kernel [Adaptada de: (“Start Here – Xenomai”)].

- Núcleo Mercury: conta com as funcionalidades de tempo real nativas do *kernel* do Linux. Geralmente requer a extensão PREEMPT-RT ativada no *kernel* para entregar serviços de tempo real. Nesta configuração unitária de *kernel* todas as APIs não-POSIX RTOS Xenomai são executadas sobre biblioteca de thread nativa. *Figura 3.5* representa configuração de *kernel* unitária Xenomai.

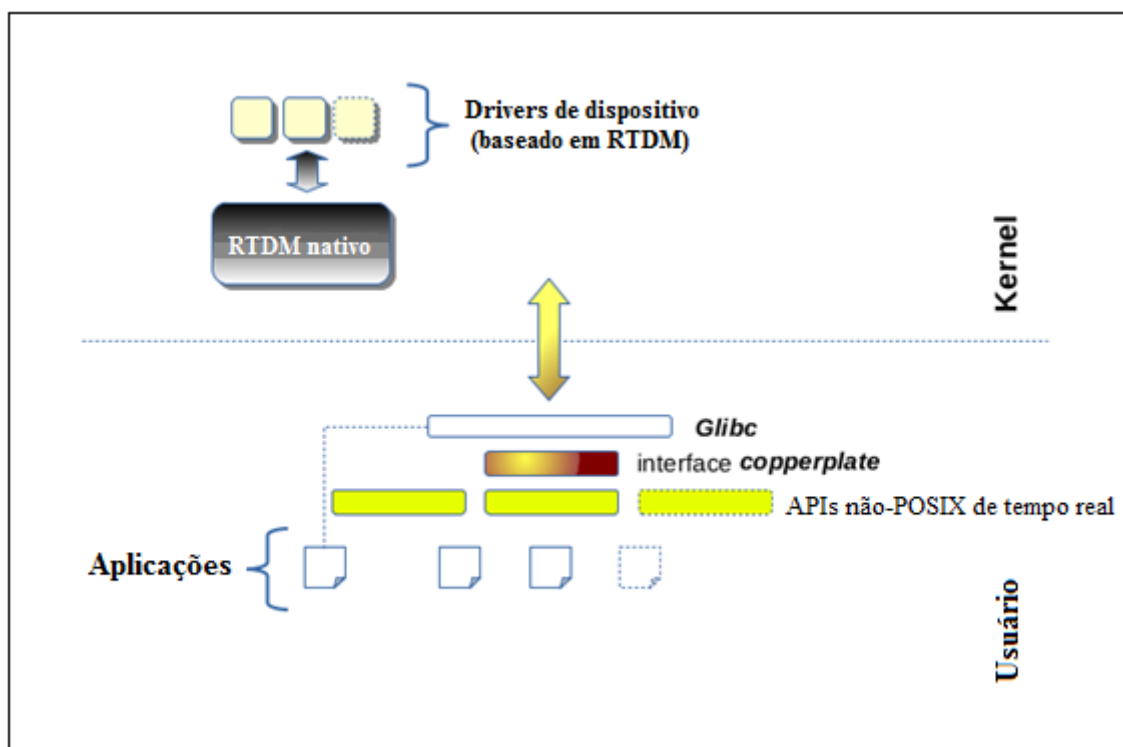
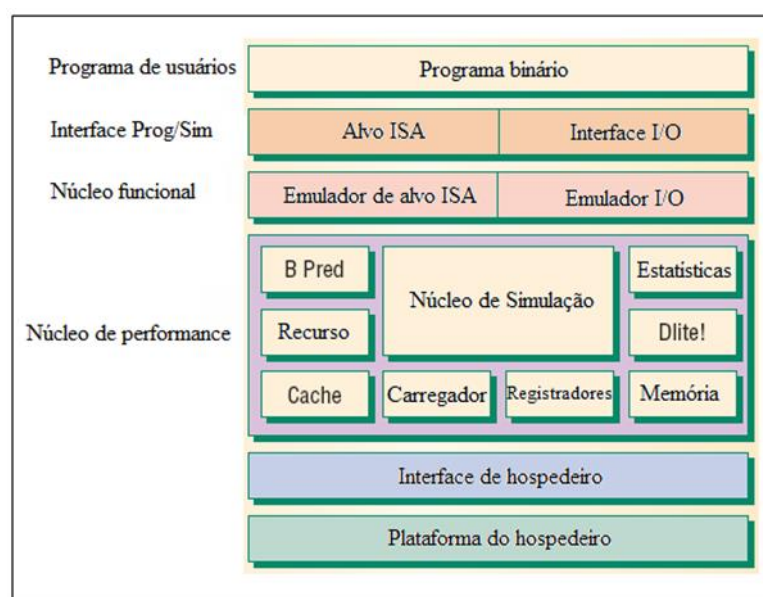


Figura 3.5 - Configuração de kernel unitária Xenomai. [Adaptada de: ("Start Here – Xenomai")].

### 3.5 Ferramentas educacionais para programação paralela

Existem diversas ferramentas que auxiliam a construção e depuração de sistemas paralelos, conforme *item 2.5*, entretanto a alta curva de aprendizado necessária para utilizá-las acaba não as tornando atrativas para principiantes. Na literatura científica poucas ferramentas existem com a abordagem específica do auxílio no aprendizado.

Em (PERISSATTO et al., 2007) é citada a utilização da ferramenta de simulação SimpleScalar que é uma ferramenta de simulação de multiprocessadores superescalares com memória compartilhada. Nessa ferramenta é possível simular uma máquina com múltiplos processadores fazendo uso do modelo de memória compartilhada. É também possível especificar parâmetros como latência de acesso a memória, tamanho e estrutura de caches, entre outras funções. Como vantagens, pode-se citar a simulação de diversas arquiteturas, programação paralela e coerência de caches. Na *Figura 3.6* temos um diagrama de blocos da arquitetura SimpleScalar.



**Figura 3.6 - Diagrama de blocos da arquitetura SimpleScalar [adaptado de: (AUSTIN et al., 2002)]**

A ferramenta StarHPC foi criada e utilizada para ensino de programação paralela no Instituto de Tecnologia de Massachusetts. A STAR (*Software Tools for Academics and Researchers*) é uma imagem de máquina virtual que contém um conjunto de pacotes, scripts e ferramentas contando ainda com a imagem da EC2 (*Amazon Elastic Computing Cloud*). A EC2 permite alugar recursos computacionais sob demanda de computadores de data center da Amazon a custos atraentes.

O principal objetivo da StarHPC é tornar fácil a utilização, o acesso e o balanceamento de carga de usuários. Normalmente é necessário o despendimento de bastante tempo na configuração do ambiente, porém com uso dessa ferramenta essa etapa é pulada, permitindo que se passe diretamente à prática e ensino. Além

disso, por usar um serviço sob demanda, é possível obter economia custos de hardware, manutenção, energia e resfriamento (IVICA et al., 2009). A *Figura 3.7* apresenta a arquitetura da solução STARHPC.

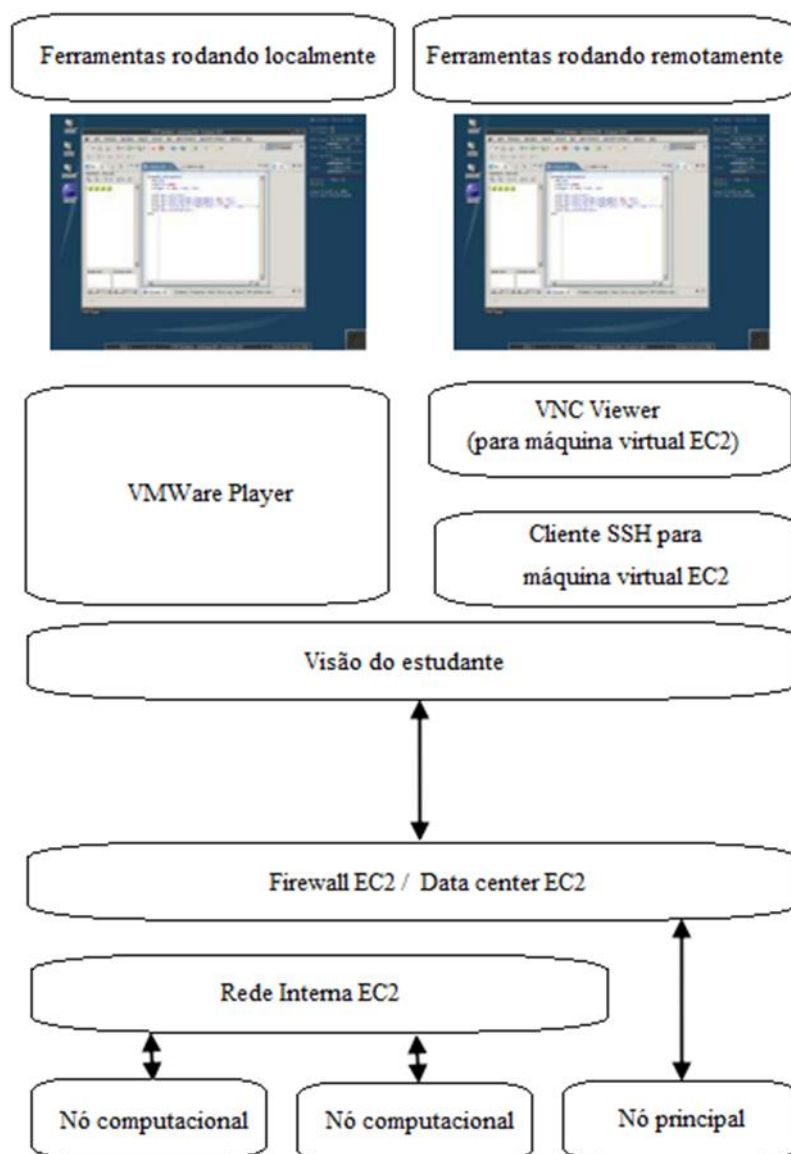


Figura 3.7 - Arquitetura da solução StarHPC [Adaptado de (IVICA et al., 2009)].

# Capítulo 4

## DISCUSSÃO DOS OBJETOS DO KERNEL RTXC

---

---

### 4.1 Responsabilidades de um kernel de tempo real

Um *kernel* de tempo real tem como responsabilidade produzir saídas a estímulos externos, em período de tempo determinístico. Além disso, o *kernel* é responsável por gerenciar o processador, recursos do sistema, atendendo os requisitos de aplicação que foi projetado. Deve ser capaz de responder e sincronizar eventos, mover dados eficientemente entre processos e gerenciar demandas de processos de acordo com as restrições de tempo.

Mantido pela Quadros Systems, o *kernel* do RTXC RTOS busca atender essas responsabilidades de tempo real, atingindo alto desempenho para sistemas embarcados, alta confiabilidade sendo consolidado há mais de vinte e cinco anos no mercado de sistemas embarcados.

### 4.2 Introdução ao kernel RTXC

O *kernel* RTXC é subdividido em três níveis de execução. A zona 1: IS, nível de interrupção, tem a maior prioridade dentre as demais, ou seja, enquanto são realizados procedimentos na zona 1, nenhuma outra zona pode operar. A zona 2: TS, nível de Thread que possui prioridade média, e é nela que muitas das operações do

RTXC/ss ocorrem. Zona 3: KS, nível de Tarefa que é tipicamente classificada como controle de funções.

Dois modos de operações são existentes no *kernel*: o framework de uma pilha RTX/ss e o de mais de uma pilha RTX/ms. Cada um destes modos apresenta escalonador específico e chamadas próprias de sistema. A *Figura 4.1* apresenta as zonas de operações do *kernel*.

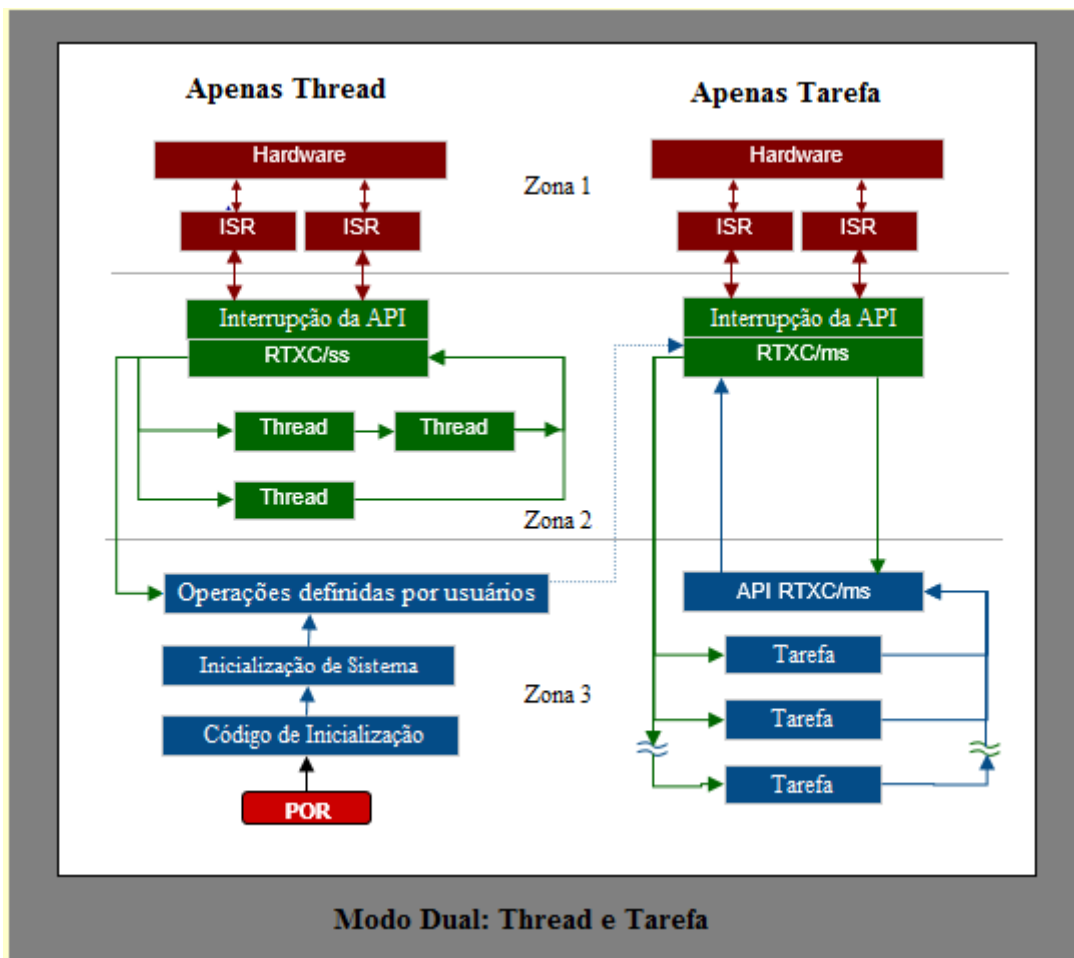


Figura 4.1 - Zonas de Operação Quadros RTX [Adaptado de: ("RTXC Quadros Manuals")]

O framework RTX/ss por ser um modelo de uma pilha, e ser constituído de Threads de baixa latência para escalonamento, é adequado para aplicações que tenham processamento de interrupção de alta frequência, como por exemplo processamento de sinal digital.

Para RTX/ms o framework proporciona um modelo de múltiplas pilhas independentes, escalonamento preemptivo multitarefa e um amplo conjunto de

serviços de *kernel* tornando-o adequado para requerimentos determinísticos e requerimentos de sistemas de tempo real hard.

Devido à alta escalabilidade do sistema tanto o modo de uma pilha (RTX/ss) quanto o de múltiplas pilhas (RTX/ms) podem ser usados separadamente ou em

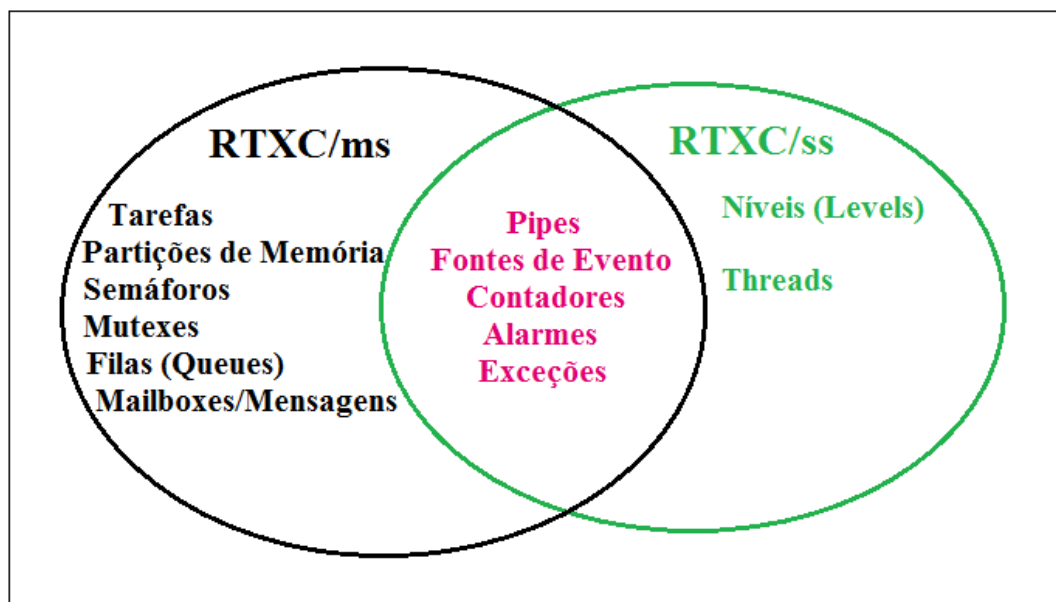


Figura 4.2 - Esquema dos componentes RTX/ms e RTX/ss [Adaptado de: (“RTX Quadros Manuais”).

conjunto para construções de aplicações. Cada um desses componentes possui serviços de *kernel* de acordo com sua característica conforme a *Figura 4.2*.

### 4.3 Níveis (*Levels*)

Nível ou *Level* é um tipo de classe especial para RTX/ss que tem como função organizar operações em sua classe filha, *Threads*. Os níveis podem variar de 1 até 16, o escalonamento da execução dos *Threads* obedece ao nível que está associado. Sendo assim, quanto menor o valor numérico de um nível, maior a sua prioridade de execução.



## 4.4 Objetos executáveis do *Kernel RTX*

Como objetos executáveis do *kernel* temos: Exceções (Zona 1), Threads (Zona 2) e Tarefas (Zona 3).

### 4.4.1 Exceção (Exception)

Um sistema embarcado precisa responder a estímulos externos geralmente este processo ocorre através de monitoramento ou gerenciamento desses eventos externos. Dispositivos conectados ou associados a esse processo necessitam que o sistema execute a ação em resposta ao estímulo. Para cada exceção ou interrupção existe uma parte de código responsável por lidar com a demanda, que é chamada ISR (*Interrupt Service Routine*- Rotina de Interrupção de Serviço).

O *kernel RTX* trata a exceção de forma generalizada através do objeto *Exception*. O objeto de exceção faz a conexão entre a fonte de interrupção e o código de aplicação que a atende. Esse objeto é associado a cada interrupção e contém propriedades que transferem diretamente a CPU a rotina de interrupção de serviço.

O principal objetivo de uma exceção como classe é possibilitar que os drivers de dispositivos sejam carregados em tempo de execução, permitindo assim que os vetores de interrupção associados aos dispositivos sejam chamados enquanto o sistema está em operação.

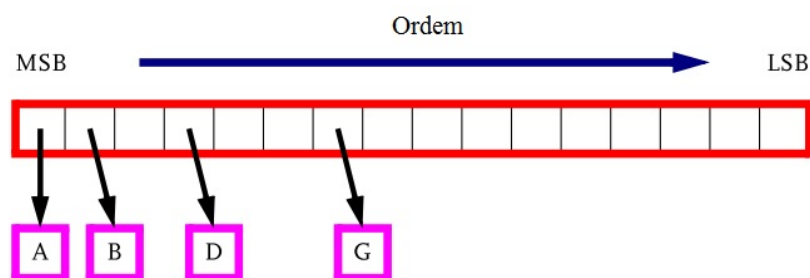
### 4.4.2 Threads

Em sistemas embarcados de tempo real, as unidades de execução podem ser decompostas em unidades menores de execução chamadas de threads. Os threads executam funções com dois parâmetros opcionais de entrada que são os ponteiros de void para: *arg* e *envarg*. No ponteiro *arg* são passados os argumentos de entrada e em *envarg* são passados argumentos de variáveis de ambiente.

Os threads são organizados de acordo com seu nível e sua ordem. Um nível com valor absoluto menor possui maior prioridade, e dentro de um mesmo nível o maior valor absoluto de ordem determina maior prioridade, sobre um valor de ordem

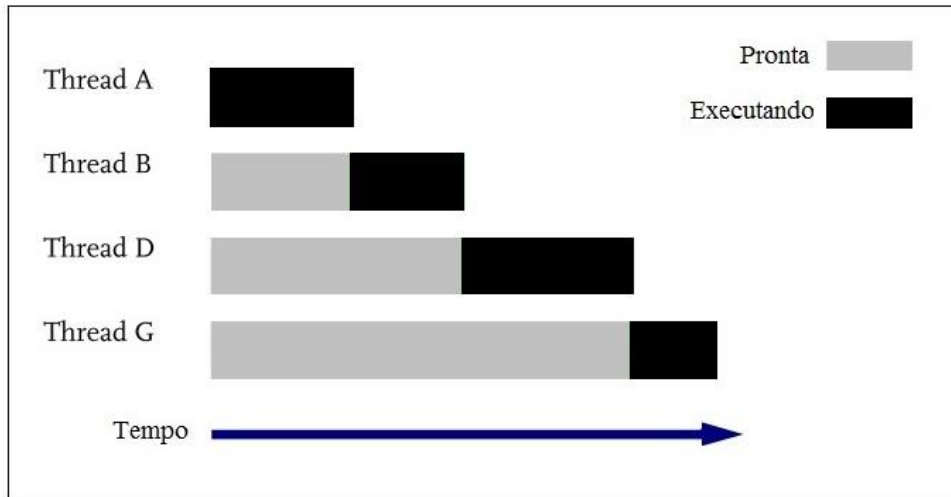
menor. Um thread não é interrompido até que todo o seu trabalho seja executado, caso outro thread de prioridade maior seja escalonado, então o thread de menor prioridade é interrompido e o de maior prioridade executa seu trabalho. Um thread de prioridade menor não é executado até que os threads de prioridade maior tenham sido executadas.

Existem duas formas de escalonamento: Round-Robin e prioridade. O escalonamento Round-Robin é um dos escalonamentos mais simples e antigos existentes: o escalonador começa fornecendo o controle da CPU para o thread atual (corrente) e em seguida percorre o próximo elemento de forma a atender todos os elementos da tabela de threads prontos para execução. De acordo com a *Figura 4.3*, têm-se o arranjo de thread A até thread G armazenados no vetor variando do bit mais significativo (MSB – *Most Significant Bit*) até o bit menos significativo (LSB – *Least Significant Bit*) nesse arranjo o Thread de maior ordem encontra-se à esquerda (Thread A) e o de menor ordem a direita (Thread G).



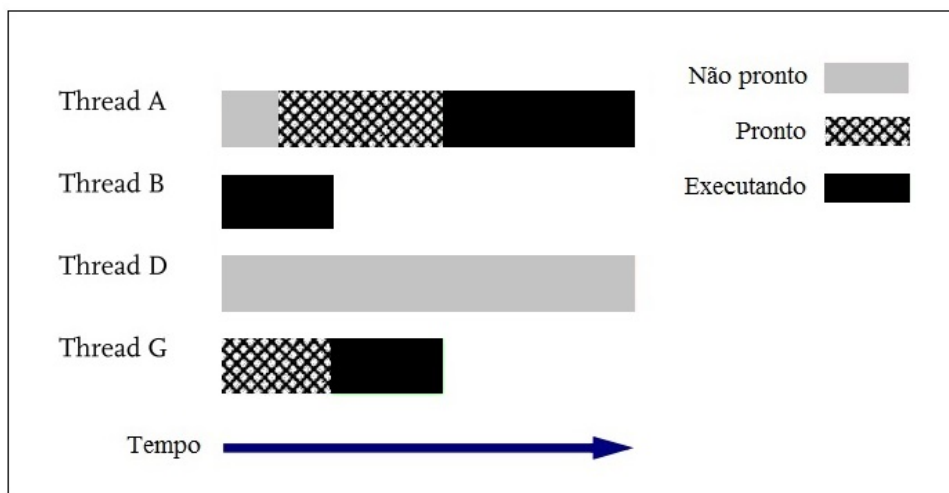
**Figura 4.3 - Exemplo de tabela de escalonamento [Adaptado de: (“RTXC Quadros Manuals”)]**

Considerando que todos os Threads estejam prontos a sequência de execução para o escalonamento Round-Robin ocorre de acordo com a *Figura 4.4*:



**Figura 4.4 - Primeiro exemplo de sequência de tempo Round-Robin. [Adaptado de: (“RTXC Quadros Manuals”)]**

Já em uma outra situação exemplo, caso apenas os threads B e G estejam prontos, o controle é dado ao thread B. Em seguida, mesmo que o thread A tenha maior prioridade que o thread G, o controle da CPU é dado ao thread G por ser o próximo pronto da lista do arranjo de threads a se encontrar no estado pronto. Conforme se observa na *Figura 4.5*, após executar o thread G, caso o thread A passe para o estado pronto, o escalonador recomeça a percorrer o arranjo a partir do MSB, executando então o thread A.



**Figura 4.5 - Segundo exemplo de sequência de tempo Round-Robin. [Adaptado de: (“RTXC Quadros Manuals”)]**

Para a política de escalonamento do Prioritário, que é a forma de escalonamento padrão para RTXC, temos a execução obedecendo aos níveis de prioridade. Para os níveis de menor valor temos maiores prioridades de execução, já dentro de um mesmo nível para valores de ordem maiores temos maiores prioridades de execução desse mesmo nível. A *Figura 4.6* mostra o escalonamento por prioridade.



**Figura 4.6 - Exemplo de escalonamento por prioridade. [Adaptado de: (“RTXC Quadros Manuals”)]**

Conforme a *Figura 4.6*, o thread B inicia a execução, depois embora tanto o thread A e G estejam prontos, o thread A é executado por possuir valor de ordem maior e consequentemente maior prioridade. Depois o Thread G é executado por ser o único thread pronto. Na sequência os threads A e D passam para o estado pronto, porém o thread A é executado por possuir prioridade maior. Durante a execução do thread A, o thread B passa para o estado pronto, após o término da execução do thread A o controle da CPU é tomado pelo thread B, deixando o thread D em espera. Após a execução do thread B, o thread D executa e os threads A e mudam para o estado pronto. Com o término da execução do thread D, o thread A executa enquanto o thread G espera o escalonamento. Terminando o thread A, finalmente o thread G executa e tem-se o fim do exemplo. Pode-se observar quem em um mesmo nível

como o desse exemplo não ocorre a preempção, mesmo com um thread de ordem mais elevada.

O thread não possui contexto como uma *Task*(Tarefa) isso se deve à natureza de componentes de pilha única de RTX/ss por isso, não pode bloquear ou esperar por algum outro processo para então continuar. Como consequência, o thread recebe o controle da CPU no ponto de entrada, sem qualquer contexto definido do processador, sendo responsabilidade do thread estabelecer na pilha do sistema quaisquer variáveis necessárias para o ciclo de execução.

Cada thread pode ter como atributo opcional o *gate*, que nada mais é que uma variável numérica e sem sinal, que permite um controle mais sofisticado do escalonamento do thread. O kernel RTX dá suporte a chamadas que alteram o *gate* e em seguida escalonam o Thread, ou que alteram e só escalonam o Thread quando o *gate* atinge certo valor proporcionando assim maior liberdade para um escalonamento mais específico.

Existe um processo especial no componente RTX/ss que é o Thread Nulo, e este não está associado a nível, porque apenas roda quando todos os Threads de todos os níveis não estão prontos. O Thread Nulo opera logicamente na Zona 3 e precisa usar a pilha do sistema para quaisquer variáveis locais, é definido pelo usuário e pode executar um loop simples ou operações complexas para a aplicação.

#### 4.4.3 Tarefa (*Task*)

Para o desenvolvimento de uma aplicação de tempo real o desenvolvedor divide os componentes de execução em unidades menores chamadas *Tasks* ou Tarefas. O *kernel* RTX/ms oferece suporte a versão multitarefa, permitindo um maior desempenho em sistemas *multicore*. Permite também que se definam numa combinação de tarefas dinâmicas e estáticas até o total de 255 tarefas. As tarefas são organizadas em um bloco de tarefas composto por duas partes: código do programa e a estrutura de dados. O *kernel* RTX trata o código de uma tarefa como uma função. A principal diferença entre uma tarefa e uma função é que uma tarefa nunca retorna a quem a chama.

As tarefas podem seguir dois modelos, o modelo executa uma vez em que o trabalho definido é executado e em seguida a tarefa é finalizada, o segundo modelo é o modelo laço em que a tarefa é executada e reexecutada indefinidamente.

Dois são os estados para uma Tarefa: o bloqueado e o pronto. No estado bloqueado não é possível se receber o controle da CPU, todas as tarefas são iniciadas por padrão neste estado, entretanto a tarefa pode passar para estado bloqueado também quando necessita de algum recurso que não está disponível ou necessita esperar por algum evento. Através de chamadas do sistema e/ou liberação de recursos do sistema, uma tarefa pode mudar para o estado pronto. Nesse estado, ela aguarda na fila de escalonamento para obter recursos da CPU e assim iniciar ou continuar sua execução. Cada tarefa possui atributos como ponteiro de função a ser executada, pilha base, tamanho de pilha base e prioridade.

No escalonamento das tarefas o *kernel* organiza as tarefas de alta prioridade de forma que estas sejam as primeiras da lista de execução. A prioridade pode variar de um até cento e vinte e seis, sendo 1 o valor da mais alta prioridade e 126 o valor de menor prioridade. A tarefa de mais alta prioridade é a que é executada no momento. Caso outra tarefa de mesma prioridade seja adicionada à lista de tarefas prontas, essa é adicionada imediatamente após a última de mesma prioridade. No momento que uma tarefa se torna bloqueada, o *kernel* a retira da lista de tarefas prontas. O *kernel* prevê a existência de uma tarefa especial chamada tarefa Nula que possui a menor prioridade possível e garante que a fila de tarefas prontas nunca fique vazia. A tarefa nula nunca pode ser bloqueada ou ter acesso a recursos como mutex, ou enviar mensagens através de *mailbox*. As três formas possíveis de escalonamento de uma tarefa são: Round-Robin, *tick-sliced* e preemptivo.

Round Robin é uma das formas mais antigas e simples de escalonamento para sistemas multitarefa no qual cada tarefa é responsável por determinar se as condições estão corretas para executar e o quanto isso pode ser feito. Caso uma tarefa venha a ser bloqueada esta necessita ceder o uso da CPU a outra. Como regra, as tarefas que selecionam o escalonamento de Round-Robin devem possuir a mesma prioridade, caso isso não seja feito, uma tarefa de prioridade menor corre o risco de nunca receber um tempo de execução na CPU.

O escalonamento *Tick-Sliced*, ou fatiamento por *ticks*, é uma variante do Round-Robin em que a tarefa recebe o controle da CPU por uma quantidade de *ticks*

pré-determinada de quantum. Essa quantidade de *ticks* pode estar associada ao tempo ou ao número da ocorrência de um evento. Ao invés de centralizar a determinação do quantum, cada tarefa que pertence ao escalonamento deve ter seu próprio *tick* quantum. A escolha adequada para o tamanho do quantum pode tornar um sistema mais responsivo e com melhor desempenho, mesmo que não seja estritamente determinístico. Entretanto, valores muito pequenos de quantum podem degradar a performance do sistema como um todo, devido ao fato do *kernel* utilizar mais vezes os recursos do sistema para efetuar o escalonamento mais frequentemente.

No escalonamento preemptivo as tarefas são escalonadas de acordo com suas prioridades. Quando uma tarefa de maior prioridade é adicionada à lista de tarefas prontas, o *kernel* interrompe a execução da tarefa corrente e passa os recursos da CPU para a tarefa de maior prioridade. Assim que a tarefa de maior prioridade termina de executar, o *kernel* devolve o controle para a tarefa que havia sido interrompida, e esta continua a executar a partir do ponto em que havia parado.

Os métodos de escalonamento podem ser utilizados mescladamente entre as tarefas, ou seja, não é obrigatório que se use a mesma forma de escalonamento entre todas as tarefas. O *kernel* é responsável por armazenar o contexto de cada tarefa, assim caso uma tarefa seja bloqueada, interrompida ou sofra preempção o sistema é capaz de retornar do ponto em que a tarefa parou.

## 4.5 Semáforo

Um sistema multitarefa deve fornecer meios flexíveis de se gerenciar a ocorrência de eventos que podem ser internos, externos, síncronos e assíncronos. O semáforo é uma estrutura que permite o acesso controlado a recursos sensíveis do sistema. Cada semáforo possui um valor de conta que é usado para se controlar o acesso a recursos restritos ou para contagem de eventos por exemplo.

A cada chamada da função teste semáforo, este é decrementado em um no valor de conta, todavia esse número nunca fica abaixo de zero. Quando se chega ao

valor zero para a conta, qualquer requisição não terá o recurso do semáforo liberado neste caso existem três chamadas de sistema: o teste não bloqueante, o bloqueado por tempo determinado e o bloqueante.

O teste não bloqueante verifica a disponibilidade do semáforo e retorna sucesso ou falha. O teste bloqueante no caso de o semáforo não ter recurso disponível, fica bloqueado por tempo indeterminado até que o semáforo receba um sinal liberando o recurso. O teste bloqueado por tempo determinado é semelhante ao teste bloqueante exceto pelo fato, que seu bloqueio segue um tempo pré-determinado de bloqueio, por isso ao terminar o tempo de espera caso nenhum sinal seja recebido é retornada falha, já para o recebimento de sinal é retornado sucesso liberando o semáforo.

## 4.6 Mutex

Para um sistema multitarefa fornecer saídas corretas é necessário um gerenciamento correto de recursos compartilhados, algumas operações devem ter o caráter atômico em sua leitura e/ou escrita.

A estrutura mutex permite o recurso de exclusão mútua, isto é, apenas uma tarefa por vez tem acesso à leitura/escrita do trecho crítico controlado pelo mutex. Caso uma segunda tarefa tente simultaneamente acessar a região crítica, não obtém a permissão podendo tentar novamente o acesso, ficar esperando por um período de tempo pelo acesso, ou esperar indefinidamente enquanto o mutex não é liberado.

O comportamento do mutex é semelhante a um semáforo de conta um, sendo chamado por alguns autores de semáforo binário. (SILBERSCHATZ et al.,2013).

## 4.7 Partição de Memória

Toda aplicação necessita de uma certa quantidade de memória para executar, no sistema RTX todo mapeamento de memória é realizado através das partições de



memória. Cada partição consiste em número determinado de blocos todos com o mesmo tamanho. Mesmo que cada partição possa especificar o tamanho de seu bloco, diferindo entre si, dentro de uma partição todos os blocos devem possuir o mesmo tamanho.

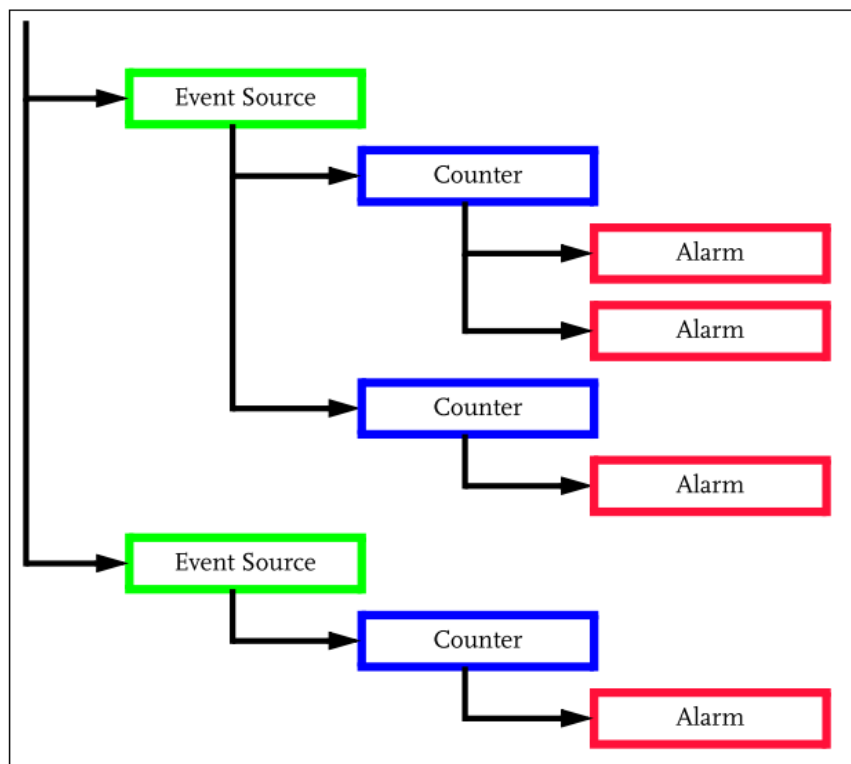
Os métodos de gerenciamento de memória podem consumir muitos ciclos ou se tornarem não determinísticos, tendo como sintomas a fragmentação de memória, situação essa que para um sistema que não seja de tempo real não pareça uma preocupação alarmante. Porém, a impossibilidade de se determinar o tempo requerido para a desfragmentação de memória pode gerar danos ao um sistema de tempo real, sendo essencial que o gerenciador seja ao mesmo tempo determinístico e previna fragmentação de memória (“RTXC Quadros Real-Time Operating System”).

A partição de memória possui dois estados: vazio e não-vazio, que remetem à disponibilidade ou não de blocos de memória. Esses estados podem ser sincronizados com um semáforo opcional, através da definição das propriedades de uma partição.

As funções de partição têm como principais recursos: obter uma partição de memória sua alocação, requisição de blocos alocados e sua devolução.

## 4.8 Fonte de Evento (*Event Source*)

Para gerenciar os eventos em RTX, são utilizadas as seguintes estruturas: *Event Source*, *Counter* e *Alarm*. Na *Figura 4.7* observa-se a estrutura hierárquica do gerenciamento de eventos.



**Figura 4.7 - Hierarquia do gerenciamento de eventos. [Extraído de: (“RTXC Quadros Manuals”)]**

Conforme pode-se observar na *Figura 4.7* *Event Source* é “pai” de *Counter* e *Counter* é “pai” de *Alarm*. Um *Event Source* pode ter mais de *Counter* associado, assim como um *Counter* pode estar associado a mais de um *Alarm*.

A classe *Event Source* fornece suporte para recurso genérico de contagem e modelo de alarme. Com auxílio dessa classe é possível o acúmulo do número de eventos e a divisão dessa contagem em acumuladores de ordem menores, permitindo melhor precisão no uso de alarmes. A fonte de eventos é imaterial podendo ser proveniente de fontes externas como uma interrupção, ou interna gerada através de uma aplicação de software.

Como forma de trabalho, o *Event Source* incrementa seu acumulador interno de acordo com o número de batidas (*ticks*) de evento. Seu propósito primário é estabelecer um arranjo de *Counters* associados a um determinado evento.

## 4.9 Contador (*Counter*)

O objeto contador (*Counter*) tem a função de contar as batidas (*ticks*) provenientes de eventos. Sua contagem ocorre de forma incremental até chegar ao valor máximo, quando então retorna para zero.

A contagem obedece a proporção da variável “*modulus*” que é definida nas propriedades de cada contador. Por exemplo, caso a proporção seja definida como dez (valor tomado arbitrariamente, para fins de exemplo) a cada dez *ticks* proveniente de eventos, o contador incrementa seu acumulador em um. O contador pode ser “pai” de um conjunto de estruturas de alarme conforme Figura 4.7(anterior) associadas a esse mesmo contador.

## 4.10 Alarm (Alarme)

Os alarmes (ALARM) são os elementos de contagem pertencentes ao nível mais baixo da hierarquia de contagem RTX. Os contadores acumulam os *ticks* provenientes de eventos e os alarmes executam ações em threads ou *Tasks* quando certos valores de contagem dos contadores são atingidos. Por isso, alarmes são medidos com relação aos valores do contador “Pai” da hierarquia de contagem.

O alarme pode ser periódico quando sua contagem é relacionada a eventos associados a tempo, ou não-periódico quando não existir tal relação. O alarme pode ser cíclico neste caso, após cada disparo, esse alarme é automaticamente rearmado e reiniciando a contagem e posterior disparo. Para um alarme não-cíclico, tem-se a passagem para o estado inativo após o disparo. A *Figura 4.8* mostra o esquema de um alarme cíclico e a *Figura 4.9* de um alarme não-cíclico.

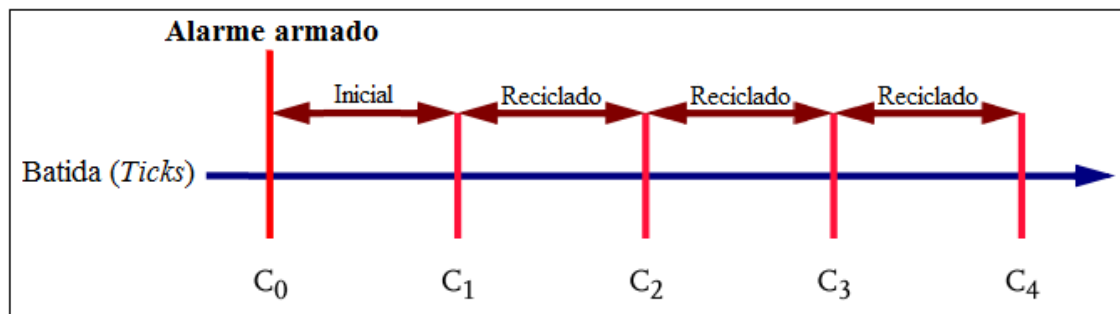


Figura 4.8 - Mostra alarme sendo armado com valor de disparo inicial de  $C_1 - C_0$  e valor disparo reciclado  $C_{n+2} - C_{n+1}$ , para  $0 \leq n \leq 2$ . [Adaptado de: (“RTXC Quadros Manuais”)]

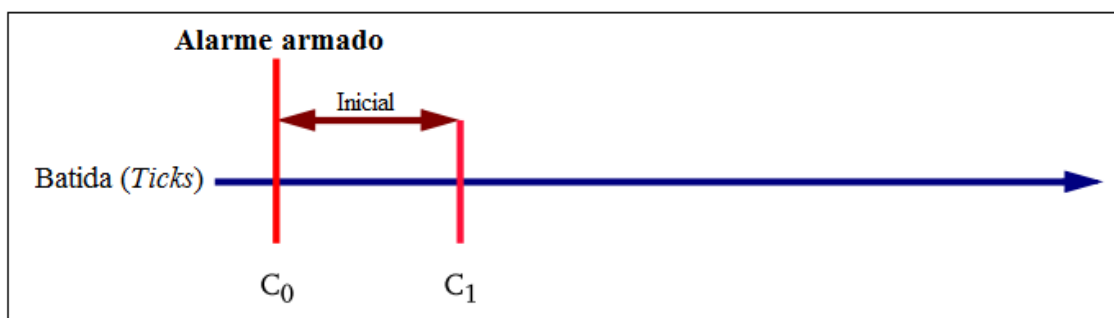


Figura 4.9 - Alarme não-cíclico, armado com valor inicial de contador  $C_0$  e disparando em  $C_1$ . [Adaptado de: (“RTXC Quadros Manuais”)]

Os contadores (*Counters*) trabalhando em conjunto com os alarmes servem a três propósitos:

- Alarmes de propósito geral: sincronizam tarefas ou escalonam Threads com eventos, após quantidade específica de *ticks* acumulada em contadores associados ao alarme.
- Alarmes internos: permitem que os serviços de kernel limitem o bloqueio de chamadas de Tarefas por um número específico de *ticks* do contador associado ao alarme.
- Contagem de *ticks* por tempo decorrido: permitem ao kernel RTX calcular o número de *ticks* que ocorrem entre dois eventos.

Para alarmes não cíclicos, principalmente no caso de estarem associados a contadores síncronos, existe a possibilidade de ocorrência de erro. Isso se deve ao fato que praticamente um alarme nunca é ativado no momento em que o contador “Pai” recebe um *tick*. A ativação do alarme vai ocorrer em algum ponto entre o último *tick* recebido pelo contador “Pai” e o próximo *tick* recebido conforme apresentado na Figura 4.10 e Figura 4.11.

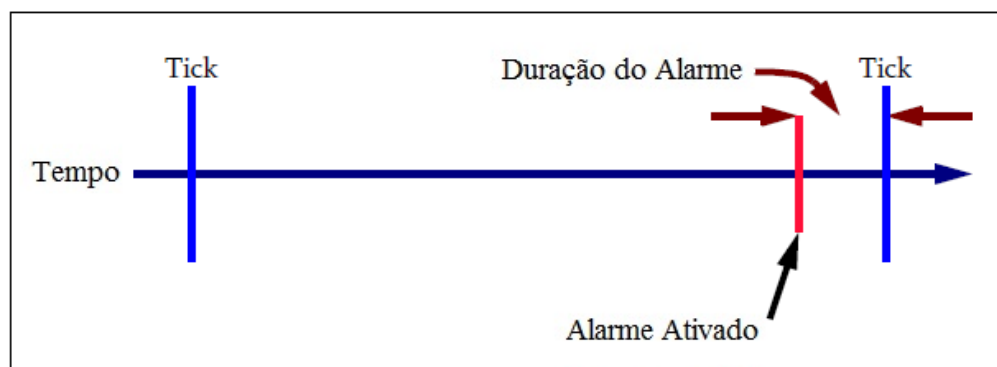


Figura 4.10 - Duração de alarme não cíclico de tempo de duração um tick, situação 1. [Adaptado de: (“RTXC Quadros Manuals”)]

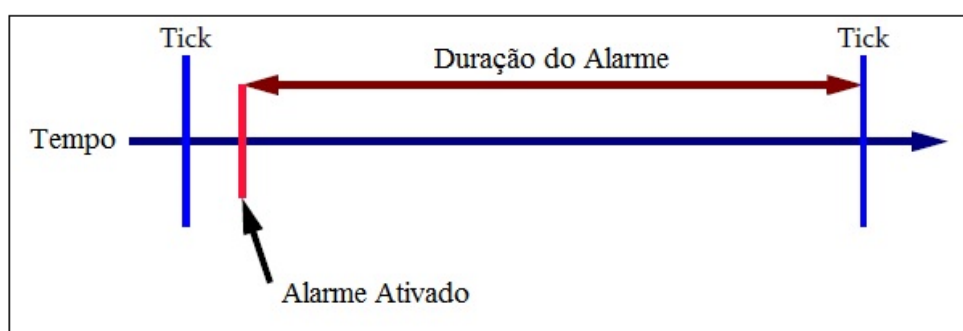


Figura 4.11 - Duração de alarme não cíclico de tempo de duração um tick, situação 2. [Adaptado de: (“RTXC Quadros Manuals”)]

O objeto alarme tem dois estados: inativo e ativo. Quando o alarme é criado ele permanece no estado inativo, quando o alarme é armado passa para o estado ativo. Caso o alarme seja não cíclico após seu disparo retorna para o estado inativo. É possível que um alarme ativo passe para o estado inativo caso sejam utilizadas chamadas do *kernel* para cancelar ou abortar.

Como propriedade opcional o alarme pode ter dois semáforos opcionais relacionados a duas ocorrências: a expiração e ao aborto do alarme respectivamente.

Quando um alarme expira uma Tarefa, de acordo com os padrões do RTXC, pode esperar, entretanto um Thread não. Existem, portanto duas ações que podem ser definidas neste caso:

- Agendar o Thread;
- Decrementar o *gate* do Thread.

Para o agendamento o efeito é direto o Thread é agendado e apenas necessita esperar pela sua vez no escalonamento para obter o controle da CPU. Se for

selecionado decrementar o *gate* do Thread, o Thread pode passar para estado pronto (espera no escalonamento) ou não de acordo com o novo valor do *gate*.

## 4.11 Fila (*Queue*)

O objeto *Queue*, ou Fila tem como função permitir cronologicamente a troca de dados entre tarefas. Portanto, esse objeto é um intermediário entre tarefas produtoras e tarefas consumidoras.

Conceitualmente, a fila é um buffer circular que contém entrada de um ou mais bytes, sendo útil para lidar com operações de fluxo de caracteres entrada e/ou saída, ou outro tipo de dado. O modelo RTX de fila permite que mais de uma tarefa insira dados na fila e mais de uma tarefa remova dados da fila.

Na fila, os dados são copiados para buffer, diferentemente de mensagens que têm apenas um ponteiro apontado para o dado. Na organização e ordenação de mensagens não existem prioridades envolvidas, sendo exclusivamente FIFO.

A fila pode ter dois estados: fila cheia, fila vazia. Por isso, dois semáforos opcionais podem ser associados a situações: fila não vazia e fila não cheia respectivamente.

## 4.12 Mailbox

O objeto *Mailbox* fornece meios para o acúmulo ordenado de mensagens, além de servir de interface entre a tarefa que envia e recebe mensagens.

O *kernel* RTX dá suporte a um número variável de *Mailboxes* que são capazes de conter mensagens de múltiplos remetentes. O *Mailbox* é tratado como objeto global e permite que múltiplas tarefas enviem mensagens ou recebam a partir de um mailbox. Entretanto, um *Mailbox* pertence somente a uma tarefa, e somente essa tarefa tem a permissão para leitura das mensagens sendo vetada a leitura de mensagens por tarefas vizinhas.

Um *Mailbox* tem dois estados: *Mailbox* vazio e não vazio. Semáforos opcionais podem estar associados ao mailbox de forma a auxiliar a sincronização de eventos relacionando aos estados.

### 4.13 Mensagem (Message)

O objeto mensagem é o componente de comunicação utilizado em *Mailbox*. Entretanto, a mensagem em si não é copiada para o destinatário, sendo enviado um envelope contendo o ponteiro de mensagem. Essa característica permite que grandes quantidades de mensagens sejam passadas com o mínimo de *overhead*. O ponteiro pode apontar tanto para uma área da memória RAM como da memória ROM, entretanto o envelope da mensagem deve estar obrigatoriamente armazenado na memória RAM.

As mensagens podem ser síncronas nesse caso o remetente aguarda uma mensagem de confirmação de recebimento do destinatário. Para mensagens assíncronas não existe a necessidade de recebimento de confirmação. As chamadas de sistema tanto de envio quanto de recebimento podem ser bloqueantes, bloqueantes por tempo determinado ou não bloqueantes. Os bloqueios podem estar associados a estados do *Mailbox*, como vazia ou não vazia, ou podem estar associados a retorno de mensagem de confirmação de recebimento.

A ordenação das mensagens em RTXC por padrão obedece a prioridade entretanto, se definido explicitamente em *Mailbox*, podem ser configuradas para FIFO (*First In, First Out*). Além disso, existe uma prioridade definida internamente com dois possíveis tipos: mensagem normal ou mensagem urgente. A mensagem normal obedece a ordenação definida nas propriedades do *Mailbox*, já a mensagem urgente é LIFO (*Last In, First Out*) ou seja, uma mensagem urgente independentemente de sua prioridade é colocada na frente das demais dentro do *Mailbox*.

## 4.14 Pipe

Para se mover dados entre entidades executando em diferentes zonas, o kernel RTX oferece o recurso da classe objeto Pipe. Esse objeto de classe permite que a partir da zona 1 o manipulador de interrupções transfira dados para um Thread (zona 2) ou Tarefa (zona 3).

O Pipe serve como um objeto intermediador fornecendo uma interface padrão entre produtor e consumidor, conceitualmente sendo um par de listas circulares. O produtor coloca dados no pipe usando o buffer e o consumidor obtém os dados do pipe como buffer de acordo com a *Figura 4.12*.

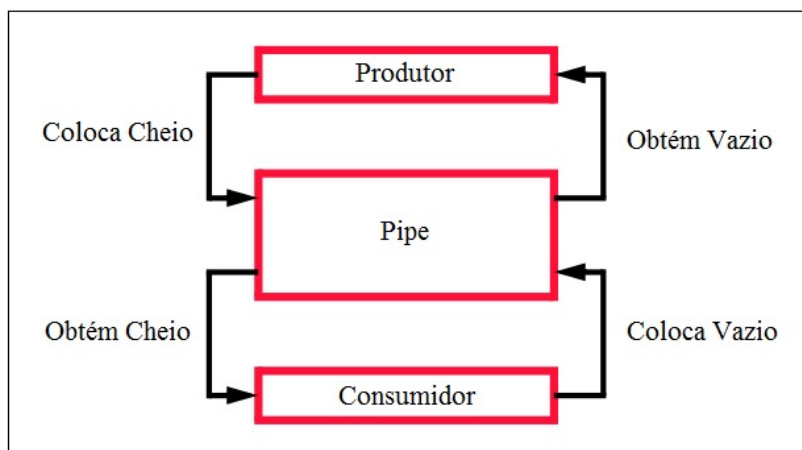


Figura 4.12 - Operações básicas do Pipe [Adaptado de: (“RTXC Quadros Manuals”)].

Os pipes são úteis para operações de *stream* de entrada/saída ou outras operações em que o uso de buffer para dados seja necessário.



# Capítulo 5

## IMPLEMENTAÇÃO DA API

---

### 5.1 Considerações iniciais

O contínuo desenvolvimento da tecnologia permitiu que a complexidade existente nos sistemas embarcados fosse capaz de executar a cadeia de desenvolvimento GNU/Linux.

O sistema operacional Linux não foi projetado para atender requisitos de tempo real, seu escalonamento padrão a partir do *kernel 2.6* baseia-se na política CFS (*Completely Fair Scheduler* - em tradução livre, Escalonador Completamente Justo), que visa a realizar a maximização do uso da CPU e desempenho na interatividade. Diferente das versões anteriores ao *kernel 2.6* que utilizavam filas de execução a CFS implementa árvore rubro-negra com o cronograma de execução das futuras tarefas, indexando os nós pelo tempo de execução em nanossegundos. O tempo máximo de execução, que é o tempo que processo aguarda para executar dividido pelo número total de processos, é calculado para cada processo. Quando o escalonador é invocado para executar um novo processo (“Inside the Linux 2.6 Completely Fair Scheduler”):

- O nó mais à esquerda da árvore de escalonamento é escolhido e enviado para execução;
- Caso o processo complete a sua execução é removido do sistema e da árvore de escalonamento;
- Caso o processo atinja o tempo máximo de execução ou é de outra forma parado este passa a ser reinserido na árvore de execução baseado no novo tempo de execução gasto;
- O nó mais à esquerda é selecionado da árvore repetindo a iteração.

Caso o processo passe muito tempo no estado *sleep*, o seu valor de tempo gasto se torna baixo aumentando a prioridade quando se torna necessária, fazendo com que essas tarefas não fiquem menos tempo no processador do que as que estão constantemente em execução.

## 5.2 Linux em Tempo Real

Para que o Linux atenda aos requisitos de tempo real existem três principais estratégias: Implementar um *kernel* extra para tratar de necessidades de tempo real, evitar usar funcionalidades do *kernel* que poderiam causar problemas de tempo real, ou modificar características do *kernel*.

### 5.2.1 Implementação de kernel extra

Para essa abordagem um novo *kernel* é criado para tratar dos requisitos de tempo real, resultando em dois ambientes ou sistemas separados. Nesse caso não é possível utilizar todas as ferramentas ou ecossistema Linux. Como exemplo podemos citar: Xenomai, RTLinux, etc. A *Figura 5.1* representa essa abordagem.

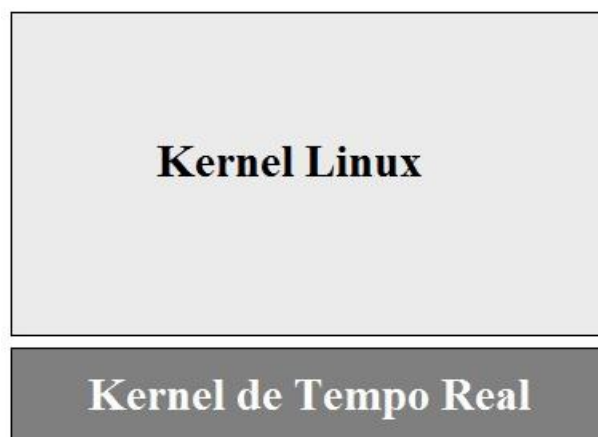


Figura 5.1 - Modelo de Kernel Dual.

### 5.2.2 Evitar usar o *kernel* para requisitos de tempo real

Para esta estratégia, o sistema é particionado em instâncias de tempo real e não tempo real. É possível configurar os processos e interrupções para rodarem com afinidade de core. São feitas menores modificações para o *kernel* threads ou *timers* para não rodarem no core de tempo real. Outra característica dessa configuração é evitar utilizar chamadas do *kernel* do Linux, buscando usar as funcionalidades do modo usuário, pois o modo de usuário é preemptível já as chamadas de *kernel* não são. A *Figura 5.2* representa essa estrutura.



Figura 5.2 - Modelo de execução espaço de usuário.

### 5.2.3 Modificar características do *kernel* do Linux

O *kernel* do Linux sofre mudanças significativas para atingir os requisitos de tempo real. Como exemplo, pode-se citar o *kernel* PREEMPT\_RT que altera diversas rotinas adicionando mudanças em mais 500 partes do código fonte e adicionando um total de 11500 novas linhas para a versão 3.0.27. A *Figura 5.3* representa a estrutura da abordagem de modificar o *kernel*.

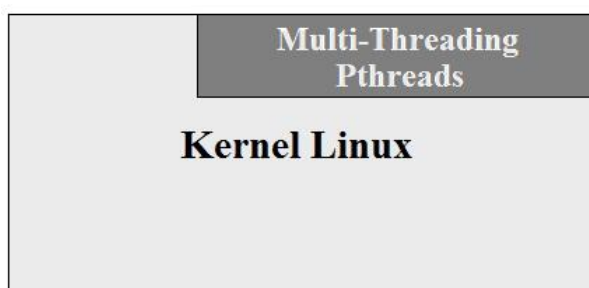


Figura 5.3 - Modelo de alteração interna do *kernel*

## 5.3 Desenvolvimento API

Os objetos e chamadas da API foram desenvolvidos em linguagem C, a abordagem usada para tornar o ambiente Linux em tempo real foi a mesma discutida no item 5.2.3 que é através de modificações no *kernel*, sendo utilizado para este trabalho o *kernel* PREEMPT\_RT juntamente com a biblioteca Pthreads. Cada uma das chamadas foi implementada de forma a manter assinatura, comportamento semelhantes ao *kernel* RTXC emulando-o para ambiente Linux. Devido a padronização POSIX para threads e sua escrita em C visar a portabilidade, é possível que se compile a API em diferentes plataformas Linux e diferentes arquiteturas gerando um código mais eficiente e otimizado.

As chamadas têm prefixo indicando a zona de funcionamento sendo IS, TS e KS correspondentes a zona 1, 2 e 3 respectivamente. O prefixo XX indica que a chamada está disponível em mais de uma zona.

### 5.3.1 Thread

Os Threads são objetos de execução para o modelo RTXC/ss de uma pilha, no desenvolvimento da API cada Thread foi mapeado para um pthread. A chamada XX\_ScheduleThread é responsável por realizar o Agendamento da execução do Thread a *Figura 5.4* mostra o pseudocódigo relativo a este procedimento. Outras chamadas implementadas são apresentadas na *Tabela 5.1*.

```
KSRC KS_ScheduleThread (THREAD thread){
    //inicializa atributos pthread de acordo com propriedades de THREAD
    inicializa_atributos_pthread();
    cria_e_inicia_agendamento_pthread();
    adiciona_TID_em_lista_encadeada_do_kernel();
    altera_estado_de_Thread_para_agendado(thread);
}
```

**Figura 5.4 - Pseudocódigo da chamada XX\_ScheduleThread.**

Tabela 5.1 - Outras chamadas do objeto Thread

Chamada	Descrição
XX_ClearThreadGateBits	Limpa os bits do <i>gate</i> do Thread
XX_DecrThreadGate	Decrementa o <i>gate</i> do Thread
XX_DefThreadArg	Define um novo ponteiro para o argumento do Thread
XX_DefThreadEntry	Define ponteiro para função do Thread
XX_DefThreadEnvArg	Define ponteiro para o argumento de ambiente do Thread
KS_DefThreadName	Define o nome de Thread dinâmico aberto anteriormente
XX_DefThreadProp	Define propriedades de Thread
TS_DisableThreadScheduler	Desabilita escalonador de Thread
TS_EnableThreadScheduler	Habilita escalonador de Thread
TS_GetThreadArg	Obtém ponteiro de argumento para o Thread
TS_GetThreadBaseLevel	Obtém o nível da prioridade base de execução do Thread
TS_GetThreadCurrentLevel	Obtém o nível de prioridade atual do Thread
XX_GetThreadEnvArg	Obtém ponteiro de argumento de ambiente para o Thread
TS_GetThreadGateLoadPreset	Obtém o valor corrente do <i>gate</i> do Thread e e seguida carrega o <i>gate</i> como o valor de predefinição
XX_GetThreadGatePreset	Obtém o valor de <i>gate</i> de predefinição
TS_GetThreadID	Obtém o manipulador do Thread corrente
KS_GetThreadName	Obtém o nome do Thread
XX_GetThreadProp	Obtém as propriedades do Thread
XX_IncrThreadGate	Incrementa o <i>gate</i> do Thread
KS_LookupThread	Busca o Thread por nome e retorna o seu manipulador
TS_LowerThreadLevel	Reduz a prioridade de execução do Thread corrente
XX_ORThreadGateBits	Define os bits do Thread <i>gate</i> usando o operador lógico OU
XX_PresetThreadGate	Define o novo valor de <i>gate</i> do Thread para o valor de <i>gate</i> de predefinição
TS_RaiseThreadLevel	Aumenta a prioridade de execução do Thread corrente
XX_ScheduleThread	Agenda a execução do Thread
XX_ScheduleThreadArg	Agenda a execução do Thread e define no argumento
XX_SetThreadGate	Define o valor do Thread <i>gate</i> e valor de predefinição
XX_SetThreadGatePreset	Define novo valor de predefinição de Thread <i>gate</i>
XX_UnscheduleThread	Desagenda a execução do Thread

### 5.3.2 Tarefa

Na implementação do objeto TASK componente RTXC/ms, cada tarefa foi mapeada para uma pthread e seus atributos inicializados de acordo com a definição das propriedades. As tarefas ativas foram adicionadas em um vetor visível para o kernel. As tarefas podem ser suspensas através da chamada KS\_SuspendTask e recomeçadas através da chamada XX\_ResumeTask a *Figura 5.5* mostra o

pseudocódigo da chamada `KS_SuspendTask` e a *Figura 5.6* da chamada `XX_ResumeTask`. Na *Tabela 5.2* outras chamadas para Tarefa estão detalhadas.

```
void KS_SuspendTask (TASK task){
    //busca id da pthread associada a Tarefa
    kernel_busca_pid_pthread_no_vetor(task);
    envia_sinal_suspendendo_pthread(tid);
}
```

**Figura 5.5 - Pseudocódigo de `KS_SuspendTask`.**

```
void XX_ResumeTask(TASK task){
    //busca id da pthread associada a Tarefa
    kernel_busca_pid_pthread_no_vetor(task);
    envia_sinal_reiniciando_pthread(tid);
}
```

**Figura 5.6 - Pseudocódigo de `XX_ResumeTask`.**

**Tabela 5.2 - Outras chamadas para objeto Tarefa.**

Chamada	Descrição
<code>KS_AbortTask</code>	Aborta a execução de uma Tarefa
<code>KS_CloseTask</code>	Encerra o uso de Tarefa dinâmica
<code>KS_DefTaskName</code>	Define o nome de uma Tarefa dinâmica aberta anteriormente
<code>KS_DefTaskPriority</code>	Define nova prioridade para Tarefa
<code>KS_DefTaskSema</code>	Associa semáforo com o término ou aborto de Tarefa
<code>KS_DefTickSlice</code>	Define o <i>quantum</i> de fatia de <i>tick</i> para a Tarefa
<code>KS_ExecuteTask</code>	Executa Tarefa
<code>KS_GetTaskID</code>	Obtém o manipulador da Tarefa corrente
<code>KS_GetTaskName</code>	Obtém o nome de uma Tarefa
<code>KS_GetTaskPriority</code>	Obtém a prioridade da Tarefa
<code>KS_GetTaskSema</code>	Obtém o semáforo associado com o término ou aborto da Tarefa
<code>KS_GetTickSlice</code>	Obtém o <i>quantum</i> de fatia de <i>tick</i> para a Tarefa
<code>KS_LookupTask</code>	Busca uma Tarefa dinâmica por nome e retorna seu manipulador
<code>KS_OpenTask</code>	Aloca e nomeia uma Tarefa dinâmica
<code>KS_SleepTask</code>	Coloca a Tarefa corrente para dormir por um período de tempo
<code>KS_TerminateTask</code>	Pára com a Tarefa removendo-a da lista de Tarefas Prontas e atribuindo status de inativa
<code>KS_UseTask</code>	Busca uma Tarefa dinâmica por nome e marca-a para uso
<code>KS_YieldTask</code>	Concede recursos para a próxima Tarefa de mesma prioridade

### 5.3.3 Fonte de eventos, Contador e Alarme

Os objetos responsáveis pelo gerenciamento de eventos são: Fonte de Eventos, Contadores e Alarmes. Na declaração e inicialização desses objetos cada um deles é associado entre si de acordo com a hierarquia, tendo ponteiros ligando pai e filho respectivamente.

Objeto fonte de eventos é responsável por processar a ocorrência de eventos atualizando o valor em seu acumulador interno e no contador

Os disparos de eventos a ser gerenciados ocorrem através da chamada `XX_ProcessEventSourceTick` que possui dois argumentos: a fonte de evento responsável pelo processamento e o número de ocorrências de eventos a ser contabilizado. O pseudocódigo correspondente na *Figura 5.7* mostra as ações realizadas a partir desta chamada. Outras chamadas do objeto Fonte de eventos são descritas na *Tabela 5.3*.

```
XX_ProcessEventSourceTick(EVNTSRC eventsrc, TICKS nevnts){
    Atualiza o acumulador da Fonte de Evento;
    Incrementa o Contador associado a Fonte de Evento;
}
```

**Figura 5.7 - Pseudocódigo da chamada `XX_ProcessEventSourceTick`.**

**Tabela 5.3 - Outras chamadas de *Event Source*.**

Chamada	Descrição
<code>XX_ClearEventSourceAttr</code>	Limpa um ou mais atributos de fonte de evento
<code>KS_CloseEventSource</code>	Finaliza o uso de fonte de evento dinâmica
<code>KS_DefEventSourceName</code>	Define o nome de uma fonte de evento aberta anteriormente
<code>XX_DefEventSourceProp</code>	Define propriedades de fonte de evento
<code>XX_GetEventSourceAcc</code>	Obtém o acumulador de fonte de evento
<code>KS_GetEventSourceName</code>	Obtém o nome de fonte de evento
<code>XX_GetEventSourceProp</code>	Obtém propriedades de fonte de evento
<code>KS_LookupEventSource</code>	Busca fonte de evento por nome retornando seu manipulador
<code>KS_OpenEventSource</code>	Aloca e nomeia uma fonte de evento dinâmica
<code>XX_SetEventSourceAcc</code>	Atribui um valor específico ao acumulador de fonte de evento
<code>XX_SetEventSourceAttr</code>	Define um ou mais atributos de fonte de evento
<code>KS_UseEventSource</code>	Busca fonte de evento por nome e marca-a para uso

O objeto Contador tem como objetivo contar batidas proveniente de eventos e armazená-las em sua estrutura interna, de acordo com a variável interna *modulus* descrita no capítulo 4 item 9. As chamadas pertencentes ao domínio dos contadores estão descritas na *Tabela 5.4*.

**Tabela 5.4 - Outras chamadas de Contador.**

Chamada	Descrição
XX_ClearCounterAttr	Limpa um ou mais atributos do contador
KS_CloseCounter	Encerra o uso de um contador dinâmico
KS_DefCounterName	Define o nome do contador dinâmico aberto anteriormente
XX_DefCounterProp	Define as propriedades do contador
XX_GetCounterAcc	Obtém o valor do acumulador de ticks do contador
KS_GetCounterName	Obtém o nome do acumulador
XX_GetCounterProp	Obtém as propriedades do contador
XX_GetElapsedCounterTicks	Computa o numero de ticks no contador ocorrido entre dois eventos
KS_LookupCounter	Procura pelo nome do contador e obtém seu manipulador
KS_OpenCounter	Aloca e nomeia um contador dinâmico
XX_SetCounterAcc	Atribui ao acumulador do contador um valor específico
XX_SetCounterAttr	Modifica um ou mais atributos do contador
KS_UseCounter	Busca um contador por nome e marca-o para uso

Os Alarmes pertencem a hierarquia mais baixa do conjunto de objetos responsáveis pela contagem de eventos. A partir do momento em que a função de iniciar o Alarme é chamada, o escalonador interno de alarmes é ativado e passa a gerenciar os disparos dos alarmes conforme pseudocódigo da *Figura 5.8*. Quando níveis de contagem pré-estabelecidos são atingidos no Contador associado ao Alarme, este dispara o escalonamento de Thread ou decreta o *gate* conforme as propriedades do Alarme.

Os principais serviços relativos ao objeto Alarme são: `KS_TestAlarm`, `KS_TestAlarmT` e `KS_TestAlarmW`.

A função `KS_TestAlarm` retorna o estado do alarme, indicando se o mesmo está inativo ou ativo assim como também o número de *ticks* restantes para o alarme em questão disparar. Na função `KS_TestAlarmT` é passado por parâmetro o número de *ticks* correspondentes ao tempo de espera, essa função retorna valores indicando se o alarme expirou antes ou depois do período de espera, além de outras ocorrências: como o alarme ser abortado, cancelado ou encontrar-se inativo antes da chamada da função. A função `KS_TestAlarmW` aguarda o alarme expirar, ser



cancelado ou abortado além de retornar sucesso ou código correspondente a interrupção juntamente com o número de *ticks* restantes para expiração do alarme.

A *Figura 5.9* apresenta um pseudocódigo da função `KS_TestAlarmW`, as demais variantes das funções de teste de alarme funcionam de forma análoga exceto pelo período de bloqueio ou ausência de bloqueio. Outras chamadas do objeto Alarme são apresentadas na *Tabela 5.5*.

```
void* escalonador_alarme(void){
    while(1){
        //Sleep thread de acordo com a resolução
        NanosleepThread(Resulacao_Alarme);
        Percorre_lista_de_Alarmes(){
            if(acumulador_contador >= disparo_de_alarme)
                adquire_mutex();
            if(alarme_ativo){
                if(alarme_action == SCHEDULETHREAD)
                    escalonar_thread_associada_ao_alarme();
                else{
                    //nesse caso decrementa o gate
                    KS_DecrThreadGate(thread);
                }
                if(existe_requisicao_teste_alarme)
                    envia_sinal_para_pthread_condicional();
                libera_mutex();
            }
            //outras sub-rotinas de escalonamento
            //responsáveis por cada estado do alarme
            (...)
        }
    }
}
```

**Figura 5.8 - Pseudocódigo de escalonador de disparos do Alarme.**

```

KSRC KS_TestAlarmW(Alarm alarm, TICKS *pticks){

    if(estado_alarme == ALARME_ATIVO){
        adquire_mutex();
        solicita_requisicao_teste_alarme();
        aguarda_bloqueada_condicao_pthread();
        libera_mutex();

        //trata todas as combinações de retorno possíveis do alarme
        (...)
        return sucesso_em_espera_ou_codigo_de_estado_alarme;
    }
    else{
        //trata todas os estados iniciais possíveis do alarme
        (...)
        return codigo_de_estado_alarme;
    }
}

```

**Figura 5.9 - Pseudocódigo de função KS\_TestAlarmW.**

**Tabela 5.5 - Outras chamadas objeto Alarme.**

Chamada	Descrição
XX_AbortAlarm	Aborta um alarme ativo
XX_ArmAlarm	Arma e inicia um alarme
XX_CancelAlarm	Faz com que um alarme ativo, se torne inativo
KS_CloseAlarm	Encerra o uso de um alarme dinâmico
XX_DefAlarmAction	Define a ação a ser tomada pelo alarme quando este expirar
XX_DefAlarmActionArm	Define a ação a ser tomada pelo alarme quando este expirar e arma o alarme
KS_DefAlarmName	Define o nome de um alarme anteriormente aberto
XX_DefAlarmProp	Define a propriedade de um alarme
KS_DefAlarmSema	Associa um semáforo com evento de alarme
KS_GetAlarmName	Obtém o nome do alarme
TS_GetAlarmProp	Obtém as propriedades do alarme
KS_GetAlarmSema	Obtém o ponteiro do semáforo associado ao evento do alarme
XX_GetAlarmTicks	Obtém o número de ticks restantes até o alarme expirar
KS_LookupAlarm	Busca pelo nome de alarme e retorna seu manipulador
KS_OpenAlarm	Aloca e nomeia um alarme dinâmico
XX_RearmAlarm	Rearma e reinicia um alarme
KS_UseAlarm	Busca um alarme dinâmico por nome e marca-o para uso

### 5.3.4 Fila (Queue)

Para o objeto do kernel QUEUE temos a troca de mensagens de forma ordenada entre tarefas. Para tal as mensagens são copiadas para a fila de destino sendo adicionadas por ordem de chegada. As chamadas `KS_PutQueueData`, `KS_PutQueueDataT` e `KS_PutQueueDataW` são responsáveis pela inserção da mensagem na fila de forma não bloqueante, bloqueante por tempo determinado e bloqueante por tempo indeterminado. O pseudocódigo da *Figura 5.10* descreve o funcionamento da chamada `KS_PutQueueDataW`.

```
void KS_PutQueueDataW (QUEUE queue, const void *psource) {
    adquiere_mutex(queue_mutex);
    while(queue->tamanho_atual == queue->profundidade_máxima)
        aguarda_sinal_de_espaco_vazio_na_pthread_condicional_da_fila();
    //copia mensagem para a Fila
    copia_mensagem_byte_a_byte(queue);
    libera_mutex(queue_mutex);
    envia_sinal_indicando_que_existe_elemento_na_fila();
}
```

**Figura 5.10 - Pseudocódigo da chamada `KS_PutQueueDataW`.**

Para resgate de mensagens de objeto QUEUE as chamadas `KS_GetQueueData`, `KS_GetQueueDataT` e `KS_GetQueueDataW` são: não bloqueante, bloqueante por tempo determinado e bloqueante por tempo indeterminado respectivamente. O pseudocódigo da função `KS_GetQueueDataW` está na *Figura 5.11*. Outras chamadas do objeto fila são apresentadas na *Tabela 5.6*.

```
void KS_GetQueueDataW (QUEUE queue, const void *psource) {
    adquiere_mutex(queue_mutex);
    while(queue->tamanho_atual == 0)
        pthread_condicional_aguarda_sinal_correspondente_a_elemento_adicionado_na_fila();
    //recebe mensagem da Fila
    recebe_mensagem_byte_a_byte(queue);
    libera_mutex(queue_mutex);
    envia_sinal_indicando_elemento_foi_retirado_da_fila();
}
```

**Figura 5.11 - Pseudocódigo da chamada `KS_GetQueueDataW`.**

Tabela 5.6 - Outras chamadas do objeto Fila.

Chamada	Descrição
KS_CloseQueue	Encerra o uso de Fila dinâmica
KS_DefQueueName	Define o nome de uma Fila dinâmica aberta anteriormente
KS_DefQueueProp	Define as propriedades de uma Fila
KS_DefQueueSema	Associa o semáforo com o evento de condição da Fila
KS_GetQueueName	Obtém o nome da Fila
KS_GetQueueProp	Obtém as propriedades da Fila
KS_GetQueueSema	Obtém o manipulador do semáforo associado com o evento da Fila
KS_LookupQueue	Busca Fila por nome e retorna seu manipulador
KS_OpenQueue	Aloca e nomeia uma Fila dinâmica
KS_UseQueue	Busca Fila por nome e marca-a para uso

### 5.3.5 Mailbox e Mensagens

Para troca de mensagens realizadas por mailbox temos dois objetos envolvidos: MBOX e MSGENV sendo respectivamente objeto referente ao mailbox e envelope de mensagens. Esse serviço de mensagens é adequado e eficiente para mensagens grandes isso se deve ao fato de que a mensagem não é copiada, mas sim o ponteiro para a mensagem gerando um *overhead* mínimo.

Os serviços de envio do mailbox podem enviar mensagens assincronamente através da chamada KS\_SendMsg, enviar mensagens sincronamente esperando a notificação de recebimento por período determinado por meio da chamada KS\_SendMsgT ou enviar mensagens sincronamente permanecendo bloqueado até que a notificação seja recebida por meio de KS\_SendMsgW. A *Figura 5.12* contém o pseudocódigo do serviço KS\_SendMsgW.

```

void KS_SendMsgW (MBOX mbox, MSGENV *pmsgenv, void *pmsgbody, MSG_PRIORITY priority){
    aloca_memoria_para_envelope();
    inicializa_valores_envelope();
    atribui_ponteiro_ao_envelope(pmsgbody);

    if(priority == URGENT_MSG){
        //Insere mensagem na frente de todas as mensagens da maibox
        insereMensagemLIFO(mbox, (*pmsgenv));
    }
    else{
        if(atributos_mailbox == FIFO){
            //Insere por ordem de chegada
            insereMensagemFIFO(mbox, (*pmsgenv));
        }
        else{
            //Insere de acordo com prioridade
            insereMensagemPrioridade(mbox, (*pmsgenv));
        }
    }
    aguarda_sinal_notificação_de_recebimento();
}

```

**Figura 5.12 - Pseudocódigo do serviço KS\_SendMsgW.**

A chamada KS\_AckMsg é responsável pela sinalização do recebimento de mensagens e seu pseudocódigo é apresentado na *Figura 5.13*.

```

void KS_AckMsg (MSGENV *pmsgenv){
    envia_sinal_de_recebimento(pmsgenv);
}

```

**Figura 5.13 - Pseudocódigo de chamada KS\_AckMsg.**

A leitura das mensagens em um mailbox é realizada pelas chamadas KS\_ReceiveMsg, KS\_ReceiveMsgT e KS\_ReceiveMsgW sendo estas chamadas não bloqueante, bloqueante por período de tempo determinado e bloqueante respectivamente. O pseudocódigo referente a chamada bloqueante é apresentado abaixo na *Figura 5.14*, a descrição de outras chamadas de Mailbox está na *Tabela 5.7* e de Mensagens está na *Tabela 5.8*.

```

void * KS_ReceiveMsgW (MBOX mbox, MSGENV **pmsgenv){
    aguarda_enquanto_mailbox_está_vazia();
    adquire_mutex(mbox_mutex);
    aloca_memoria(*pmsgenv);
    //recebe envelope de maibox retirando-o em seguida
    recebe_envelope_mailbox(*pmsgenv);
    deleta_envelope_lido(mbox);
    libera_mutex(mbox_mutex);
    muda_status_envelope_para_lido(*pmsgenv);
    //retorna ponteiro para mensagem
    return (**pmsgenv)->item;
}

```

**Figura 5.14 - Pseudocódigo de ReceiveMsgW.**

Tabela 5.8 - Outras chamadas do objeto Mailbox.

Chamada	Descrição
KS_CloseMbox	Encerra o uso de Mailbox dinâmico
KS_DefMboxName	Define o nome de um Mailbox dinâmico aberto anteriormente
KS_DefMboxProp	Define as propriedades de um Mailbox
KS_DefMboxSema	Associa um semáforo com o evento Mailbox_Não_Vazio
KS_GetMboxName	Obtém o nome de um Mailbox
KS_GetMboxProp	Obtém as propriedades de um Mailbox
KS_GetMboxSema	Obtém o manipulador do semáforo associado ao evento Mailbox_não_vazio
KS_LookupMbox	Busca por nome um Mailbox e retorna seu manipulador
KS_OpenMbox	Aloca e nomeia um Mailbox dinâmico
KS_UseMbox	Busca um mailbox por nome e marca-o para uso

Tabela 5.7 - Outras chamadas do objeto Mensagem.

Chamada	Descrição
KS_AckMsg	Notifica recebimento de mensagem
KS_ForwardMsg	Encaminha mensagem a uma mailbox assincronamente
KS_TestAck	Testa recebimento de notificação de mensagem
KS_TestAckT	Testa recebimento de notificação de mensagem esperando por um número especificado de ticks
KS_TestAckW	Testa recebimento de notificação de mensagem, esperando bloqueada por essa notificação

### 5.3.6 Partição de Memória

O gerenciamento de serviços de Partição de memória é realizado pelo objeto PART sendo responsável pela alocação, fornecimento e devolução de blocos de memória.

No caso de requisição de memória, enquanto existirem blocos de memória disponíveis as solicitações são prontamente atendidas. Entretanto, quando não existirem mais blocos de memória uma fila é inicializada e o gerenciador de filas é chamado para atender e organizar as requisições, o funcionamento do gerenciador de filas é descrito no pseudocódigo da *Figura 5.15*.

```

void* gerenciadorFila(void* entrada){
    //o gerenciador recebe como entrada a Particao que deve gerenciar
    PART aux=(PART)entrada;
    while(particao_esta_em_uso){
        adquire_mutex(fila_ativada);
        aguarda_sinal_de_condicao_de_chamada_gerenciador();
        while((começo_da_fila != vazio) && particao_esta_em_uso){
            adquire_mutex(fila_de_espera);
            aguarda_sinal_de_semaforo_que_gerencia_blocos_memoria();
            obtem_endereço_de_bloco_memoria_livre();
            atribui_bloco_livre_ao_primeiro_elemento_da_fila();
            sinaliza_semaforo_de_espera_do_elemento_atendido();
            retira_da_lista_de_espera_o_elemento_atendido();
            libera_mutex(fila_de_espera);
        }
        libera_mutex(fila_ativada);
    }
}

```

**Figura 5.15 - Pseudocódigo gerenciador de filas de espera Partição de Memória.**

Como funções de requisição de blocos de memória têm-se: `XX_AllocBlk`, `KS_AllocBlkT`, `KS_AllocBlkW` sendo requisição não bloqueante, bloqueante por tempo definido e bloqueante respectivamente. As duas primeiras funções retornam um ponteiro de endereço de memória em caso de sucesso ou um ponteiro NULL em caso de falha, a terceira fica bloqueada indefinidamente aguardando a liberação de um bloco de memória. A *Figura 5.16* apresenta o pseudocódigo do serviço `KS_AllocBlkW`.

```

void* KS_AllocBlkW(PART part){
    faz_requisicao_nao_bloqueante_ao_semaforo_gerenciador_de_bloco_de_memoria();
    if(requisicao_de_bloco == sucesso)
        return bloco_de_memoria;
    else{
        declara_e_aloca_bloco_de_espera_em_fila();
        if(atributos_de_particao == FIFO)
            insere_FIFO_bloco_de_espera();
        else
            insere_por_Prioridade_bloco_de_espera();
    }
    chama_gerenciador_de_fila_enviando_sinal();
    aguarda_bloqueado_por_sinal_de_bloco_de_memoria_disponivel();
    return bloco_livre_recebido;
}

```

**Figura 5.16 - Pseudocódigo do serviço `KS_AllocBlkW`.**

Para a devolução de blocos que não serão mais utilizados temos a função `KS_FreeBlk`. O pseudocódigo das operações realizadas nesta função está na *Figura 5.17*.

```

void KS_FreeBlk(PART part, void *pblk) {
    adquire_mutex(blocos_livres);
    insere_bloco_em_lista_de_blocos_disponiveis();
    //atualiza conta de semaforo gerenciador de recursos
    atualiza_informacao_no_semaforo();
    libera_mutex(blocos_livres);
}

```

**Figura 5.17 - Pseudocódigo da função KS\_FreeBlk.**

Outras chamadas para o objeto Partição estão detalhadas na *Tabela 5.9*.

**Tabela 5.9 - Outras chamadas do objeto Partição.**

Chamada	Descrição
KS_ClosePart	Encerra o uso de uma Partição dinâmica
KS_DefPartName	Define o nome de uma Partição de memória dinâmica aberta anteriormente
KS_DefPartProp	Define as propriedades de uma Partição
KS_DefPartSema	Associa um semáforo ao evento Partição_Não_Vazia
KS_GetFreeBlkCount	Obtém o número de blocos livres na Partição
KS_GetPartName	Obtém o nome da Partição
KS_GetPartProp	Obtém as propriedades da Partição
KS_GetPartSema	Obtém o semáforo associado com o evento Partição_Não_Vazia
KS_LookupPart	Busca uma partição por nome e retorna seu manipulador
KS_OpenPart	Aloca e nomeia uma partição dinâmica
KS_UsePart	Busca uma partição por nome e marca-a para uso

### 5.3.7 Exceção

O objeto Exceção é responsável por tratar as interrupções do sistema, as chamadas do objeto Exceção implementadas estão detalhadas na *Tabela 5.10*

**Tabela 5.10 - Chamadas referentes ao objeto Exceção.**

Chamada	Descrição
KS_CloseException	Encerra o uso de exceção dinâmica
KS_DefExceptionName	Define o nome de uma exceção anteriormente aberta
XX_DefExceptionProp	Define as propriedades de um exceção
KS_GetExceptionName	Obtém o nome de uma exceção
XX_GetExceptionProp	Obtém as propriedades de uma exceção
KS_LookupException	Busca uma exceção por nome e retorna seu manipulador
KS_OpenException	Aloca e nomeia uma exceção dinâmica
KS_UseException	Busca uma exceção por nome e marca-a para uso



### 5.3.8 Mutex

O objeto Mutex responsável pelo gerenciamento de exclusão mútua foi implementado utilizando o recurso `pthread_mutex` adaptando-se os recursos existentes e adicionando recursos e propriedades características do RTXC. A *Tabela 5.11* apresenta as chamadas implementadas e comportamento.

**Tabela 5.11 - Chamadas referentes ao objeto Mutex.**

Chamada	Descrição
KS_CloseMutex	Encerra o uso de Mutex dinâmico
KS_DefMutexName	Define o nome de Mutex aberto anteriormente
KS_DefMutexProp	Define propriedades do Mutex
KS_DefMutexSema	Associa o semáforo com o evento <code>Mutex_Não_Ocupado</code>
KS_GetMutexName	Obtém nome de Mutex
KS_GetMutexOwner	Obtém a quem pertence o Mutex
KS_GetMutexProp	Obtém as propriedades do Mutex
KS_GetMutexSema	Obtém o manipulador de semáforo associado ao evento <code>Mutex_Não_Ocupado</code>
KS_LookupMutex	Busca Mutex por nome e retorna o seu manipulador
KS_OpenMutex	Aloca e nomeia um Mutex dinâmico
KS_ReleaseMutex	Libera o Mutex
KS_TestMutex	Testa a disponibilidade do mutex e caso não esteja sendo utilizado, obtém seu uso
KS_TestMutexT	Testa a disponibilidade do mutex. Caso esteja sendo utilizado, aguarda um número especificado de ticks e quando o Mutex se torna disponível utiliza-o
KS_TestMutexW	Testa a disponibilidade do mutex. Caso esteja sendo utilizado, aguarda até que o Mutex se torne disponível e então utiliza-o
KS_UseMutex	Busca Mutex por nome e marca-o para uso

### 5.3.9 Semáforo

O objeto Semáforo foi implementado utilizando e adaptando os recursos da biblioteca `semaphore.h` para o contexto e propriedades do objeto Semáforo RTXC. As chamadas do objeto Semáforo implementadas e sua descrição estão na *Tabela 5.12*.

Tabela 5.12 - Chamadas referentes ao objeto Semáforo.

Chamada	Descrição
KS_CloseSema	Encerra o semáforo dinâmico
KS_DefSemaCount	Define a conta do semáforo
KS_DefSemaName	Define o nome de um semáforo dinâmico aberto anteriormente
KS_DefSemaProp	Define as propriedades do semáforo
KS_GetSemaCount	Obtém o valor de conta atual do semáforo
KS_GetSemaName	Obtém o nome de um semáforo
KS_GetSemaProp	Obtém as propriedades do semáforo
KS_LookupSema	Busca semáforo por nome e retorna seu manipulador
KS_OpenSema	Aloca e nomeia um semáforo dinâmico
XX_SignalSema	Sinaliza semáforo
XX_SignalSemaM	Sinaliza múltiplos semáforos
KS_TestSema	Testa semáforo, versão não bloqueante
KS_TestSemaT	Testa semáforo, versão bloqueante por tempo determinado
KS_TestSemaW	Testa semáforo, versão bloqueante
KS_TestSemaM	Testa uma lista de semáforos
KS_TestSemaMT	Testa uma lista de semáforos e aguarda numero específico de ticks por sinal
KS_TestSemaMW	Testa uma lista de semáforos e aguarda por sinal
KS_UseSema	Busca semáforo dinâmico por nome e marca-o para uso

# Capítulo 6

## BENCHMARKS E TESTE NA API-RTXC-LINUX

---

---

### 6.1 Introdução benchmark

A API-RTXC-Linux foi escrita na linguagem C utilizando o padrão POSIX para Threads (Pthreads), como proposta do trabalho de dissertação do mestrado. Foi gerada uma biblioteca com chamadas e serviços de kernel RTXC, sendo recompilada para cada uma das arquiteturas em que o teste foi realizado.

Para se verificar as características de desempenho da API desenvolvida para o contexto de tempo real, foram realizados testes das principais funções implementadas nos processadores AMD, ARM e Intel a descrição detalhada está na *Tabela 6.1*, para ARM PREEMPT\_RT, entretanto foi utilizada uma versão anterior de distribuição Ubuntu em relação as demais devido ao fato de ser a última distribuição otimizada para o *kernel* PREEMPT\_RT entregue pelo fabricante.

**Tabela 6.1 - Descrição dos sistemas em que os benchmarks e testes foram realizados**

Rótulo	Processador				RAM (GB)	S. O		
	Modelo	clock (GHz)	núcleos	threads		Distribuição	kernel	versão
AMD	AMD Triple Core N830	2,1	3	3	5	Ubuntu 14.04	4.4.0-66-generic	64 bits
Intel	Intel I7 - 3635QM	2,1 ~ 3,4	4	8	16	Ubuntu 14.04	4.4.0-66-generic	64 bits
ARM - Não PREEMPT_RT	ARM Cortex - A9	1	2	2	1	Ubuntu 14.04	3.18.5-armv7-x2	32 bits
ARM PREEMPT_RT	ARM Cortex - A9	1	2	2	1	Ubuntu 12.04	3.5.0-213-omap4 PREEMPT	32 bits

Os principais recursos de um sistema de tempo real foram medidos tais como: Preempção de uma tarefa, requisição-devolução de bloco em partição de memória, troca de mensagens em Mailbox, inserção-remoção em fila (FIFO), disparo de alarme, obtenção-liberação mutex e resposta do semáforo. Cada teste foi realizado mil vezes a partir desses dados foi determinada a média, desvio padrão, valor máximo, mínimo e o coeficiente de variação para se determinar a homogeneidade das amostras de acordo com a equação:

$$\text{Coeficiente de variação} = \frac{\text{desvio padrão}}{\text{média}} \times 100 \Rightarrow CV = \frac{\sigma(x)}{x} \times 100$$

Para o coeficiente de variação, amostras com coeficiente menor ou igual a 20% são consideradas homogêneas, já para valores maiores que 20% são consideradas heterogêneas.

### 6.1.1 Preempção de tarefa

Neste benchmark foi iniciada a Tarefa 1 com prioridade máxima e a Tarefa 2 com prioridade alta menor em uma unidade que a Tarefa 1. A Tarefa 1 executa o loop, enquanto a Tarefa 2 suspende a Tarefa1 e depois a reinicia, conforme o pseudocódigo da *Figura 6.1*. Na implementação da API a tarefa é transformada em uma LWP (*Lightweight Processes*), no caso utilizando o padrão POSIX trata-se de um thread. Como se trata de um Pthread sua preempção ocorre mais rapidamente que a preempção de um processo, e essa é efetuada através de sinal enviado a pthread, conforme descrito no subitem 2 do capítulo 5.

<pre>void Tarefa1() {     Insere período e torna periódico;      for(i=0;i&lt;1000;i++)     {         Inicia medida de tempo;         Suspende Tarefa1;         Finaliza medida de tempo;     } }</pre>	<pre>void Tarefa2() {     Insere período e torna periódico;      for(i=0;i&lt;1000;i++)     {         Inicia medida de tempo;         Recomeça Tarefa1;         Finaliza medida de tempo;     } }</pre>
---	---

Figura 6.1 - Pseudocódigo para o benchmark de preempção de tarefa.

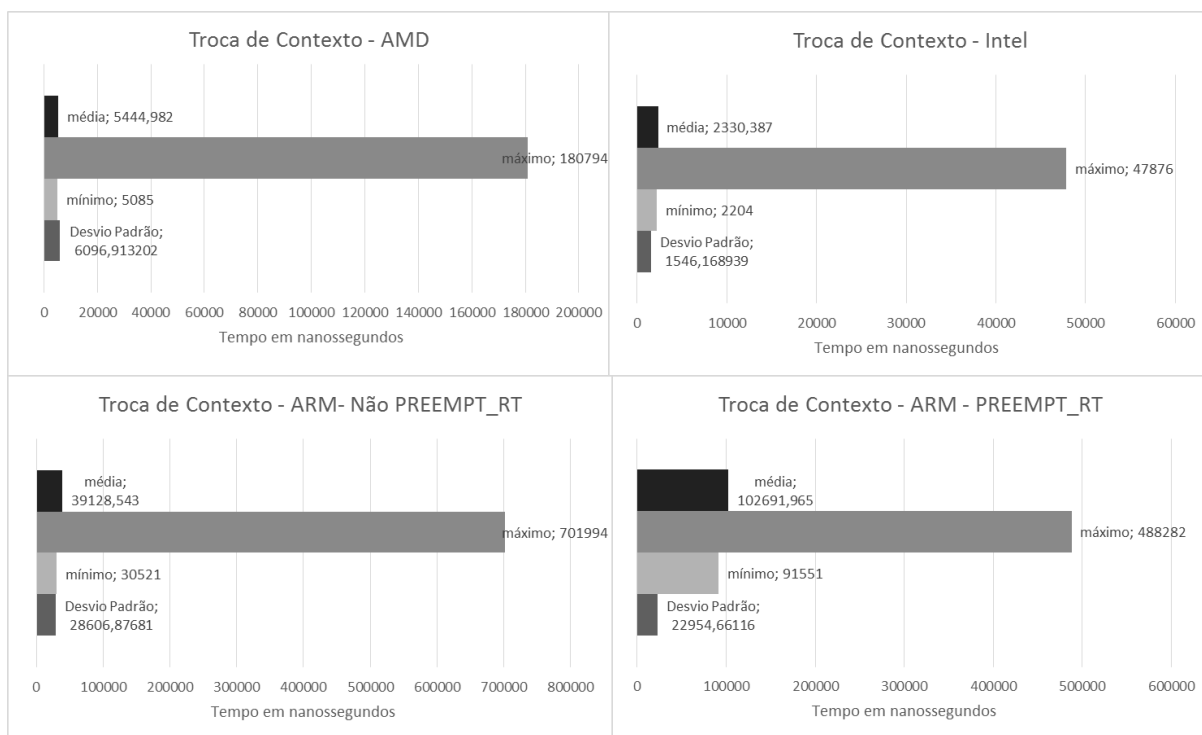


Figura 6.2 - Resultados para o benchmark de preempção de tarefa.

Tabela 6.2 - Coeficiente de variação de preempção de tarefa.

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREEMPT_RT
111,97	66,35	73,11	22,35

Para esse primeiro benchmark, conforme resultados na *Figura 6.2 e Tabela 6.2* observa-se que para ARM PREEMPT\_RT o coeficiente de variação é 22,35% valor próximo de uma distribuição homogênea, enquanto os demais valores tendem a valores de distribuições heterogêneas.

### 6.1.2 Requisição/devolução de blocos em partição de memória

O *kernel* RTXC providencia recursos para alocação de memória, que são gerenciados através do objeto Partição. O objeto partição através da definição de sua estrutura de propriedades aloca a quantidade de bytes especificada pelo usuário, que são subdivididos em pedaços menores chamados bloco de memória. A requisição pode ser feita de três formas: através de uma chamada bloqueante, bloqueante por tempo determinado e não-bloqueante. As formas bloqueantes aguardam a

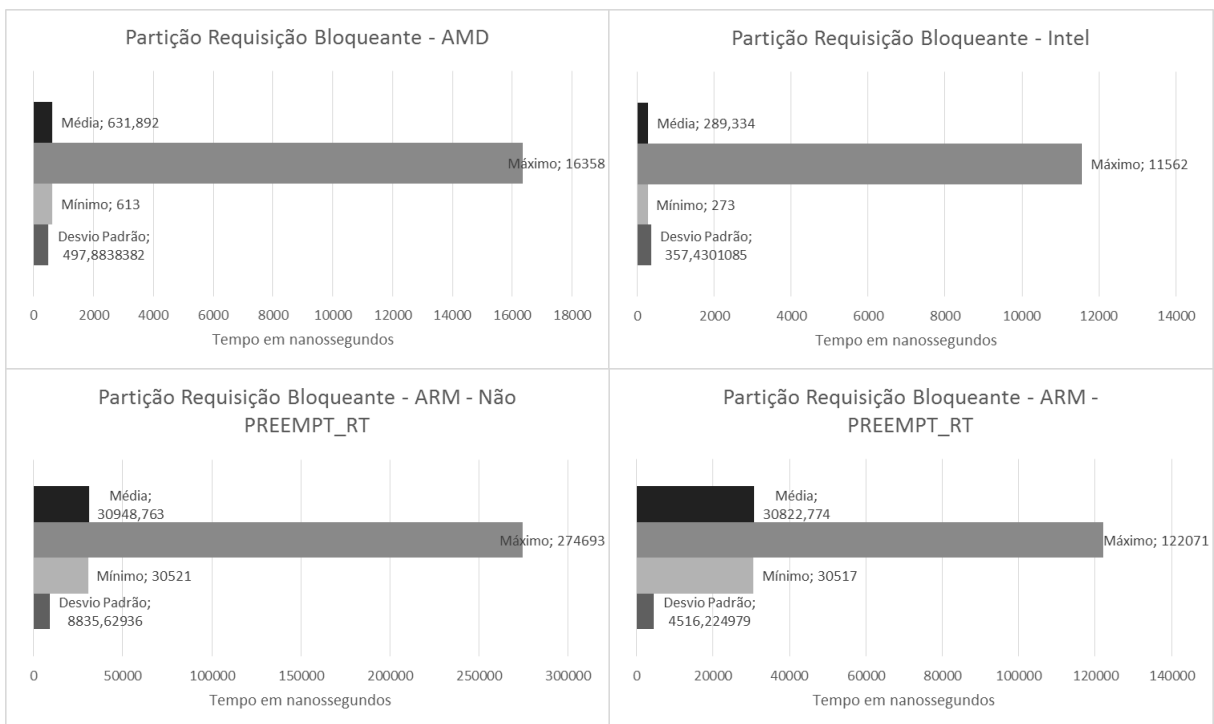
disponibilidade do bloco de memória a ser utilizado, a não bloqueante tenta obter o bloco imediatamente indicando seu sucesso ou falha.

A *Figura 6.3* mostra o pseudocódigo do teste realizado para forma bloqueante e na *Figura 6.4* temos os resultados para a versão bloqueante. O resultado para forma bloqueante por tempo determinado e não bloqueante encontra-se nos apêndices.

```
void Tarefa1()
{
    Insere um período e torna periódico;

    for(i=0;i<1000;i++)
    {
        Inicia medida de tempo;
        Requisita bloco de memória;
        Libera bloco de memória;
        Finaliza a medida de tempo;
    }
}
```

**Figura 6.3 - Pseudocódigo para o benchmark da participação de memória.**



**Figura 6.4 - Gráficos de resultados requisição/devolução bloco de memória em Partição.**

**Tabela 6.3 - Coeficiente de variação de requisição/devolução bloco de memória em Partição.**

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREEMPT_RT
78,79	123,54	28,55	14,65

A partir da *Tabela 6.3* observa-se que o coeficiente de variação de ARM PREEMPT\_RT caracteriza uma dispersão homogênea enquanto os demais caracterizam dispersão heterogênea.

### 6.1.3 Mailbox

As mensagens entre tarefas também podem ser enviadas através de Mailbox, podendo ser ordenadas por prioridade, através da ordem de chegada ou por urgência. As mensagens de Mailbox são adequadas para a comunicação de grande quantidade de dados, uma vez que nenhum dado é copiado, sendo passado apenas um envelope com o ponteiro para o conteúdo da mensagem. Para este benchmark, foram realizadas duas modalidades: o envio de mensagem síncrona e mensagem assíncrona. Na *Figura 6.5* temos o pseudocódigo para o benchmark realizado para mensagem síncrona.

```
void Tarefa1()
{
    Insere um período e torna periódico;

    for(i=0;i<1000;i++)
    {
        Inicia medida de tempo;
        Envia mensagem prioridade normal;
        Aguarda notificação de recebimento de mensagem;
        Finaliza a medida de tempo;
    }
}

void Tarefa2()
{
    for(i=0;i<1000;i++)
    {
        Aguarda recebimento de mensagem;
        Envia confirmação de chegada a Tarefa1;
    }
}
```

**Figura 6.5 - Pseudocódigo para o benchmark do mailbox bloqueante.**

Na modalidade de envio de mensagem síncrona foram testados os envios de mensagens bloqueantes utilizando duas tarefas em um dos testes as tarefas foram executadas em um mesmo *core* e no outro em *cores* diferentes. Os resultados estão na *Figura 6.6* e os demais variantes do teste encontra-se nos apêndices.

Para o benchmark realizado em apenas um core, os resultados para coeficiente de variação para AMD, Intel, ARM e ARM PREEMPT foram: 63,23%, 57,05% e 46,40% e 13,32% (distribuição homogênea) respectivamente conforme *Tabela 6.4*.

Para o benchmark realizado em dois cores, coeficiente de variação para AMD, Intel, ARM e ARM PREEMPT foram: 22,91%, 141,46% e 48,06% e 41,03% respectivamente conforme *Tabela 6.5*. No caso da divisão de tarefas em dois cores não necessariamente representa em ganho de desempenho (*speedup*), uma vez que a troca de mensagens não envolve a execução de rotinas independentes. Essas rotinas são altamente dependentes de sincronização e ganho/perda está relacionado a características de arquitetura como velocidade de cache e compartilhamento desses caches entre os núcleos do processador, além do tipo de *kernel* (Linux convencional, PREEMPT\_RT).

**Tabela 6.4 - Coeficiente de variação para Mailbox bloqueante (Um núcleo).**

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREEMPT_RT
63,23	57,05	46,40	13,32

**Tabela 6.5 - Coeficiente de variação para Mailbox bloqueante (Dois núcleos).**

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREEMPT_RT
40,46	141,46	48,06	22,91



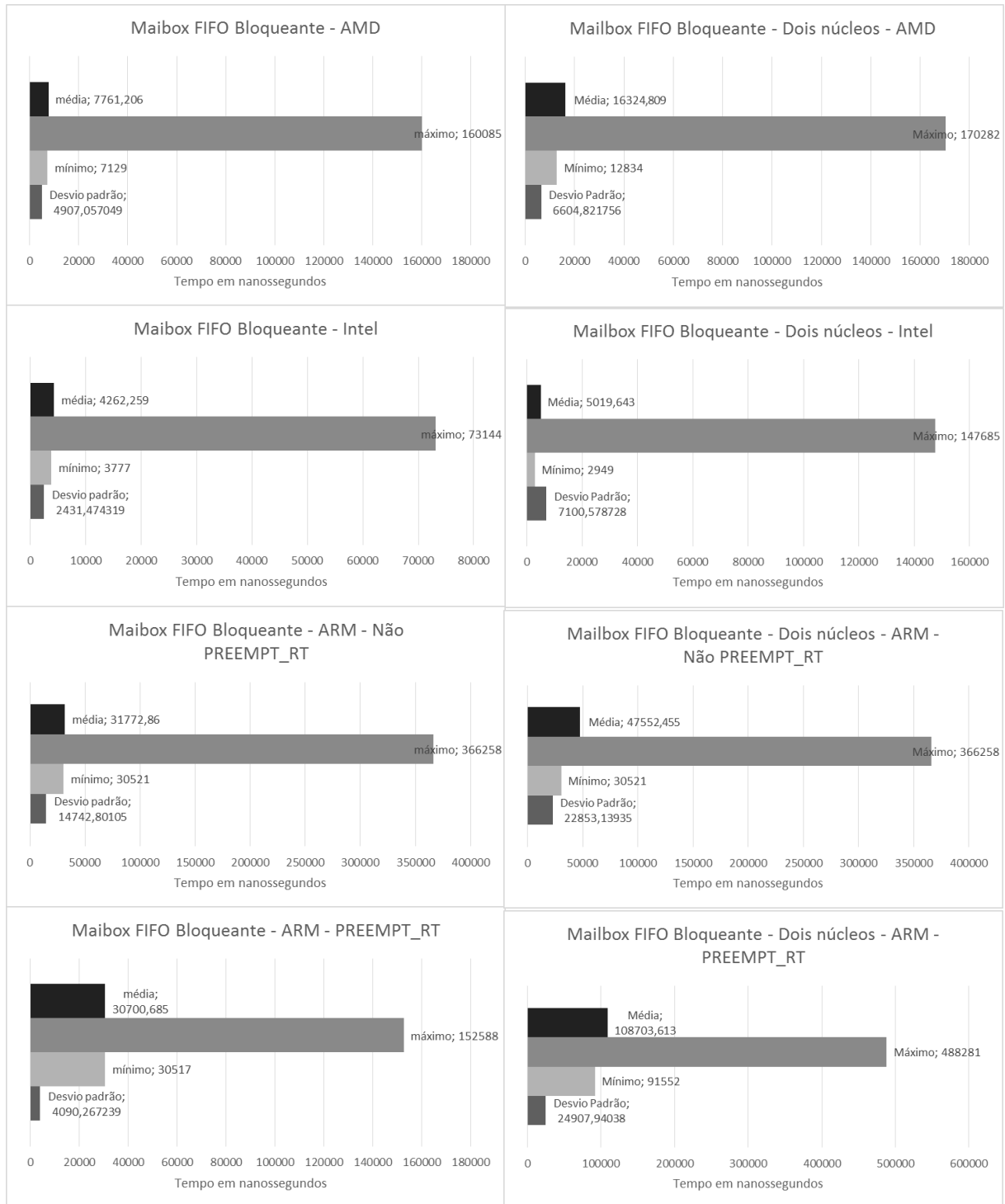


Figura 6.6 - Resultados para o benchmark de Mailbox bloqueante.

```
void Tarefa1()
{
    Insere um período e torna periódico;

    for(i=0;i<1000;i++)
    {
        Inicia medida de tempo;
        Envia mensagem prioridade normal;
        Recebe a mensagem;
        Finaliza a medida de tempo;
    }
}
```

Figura 6.8 - Pseudocódigo para o benchmark do mailbox não bloqueante.

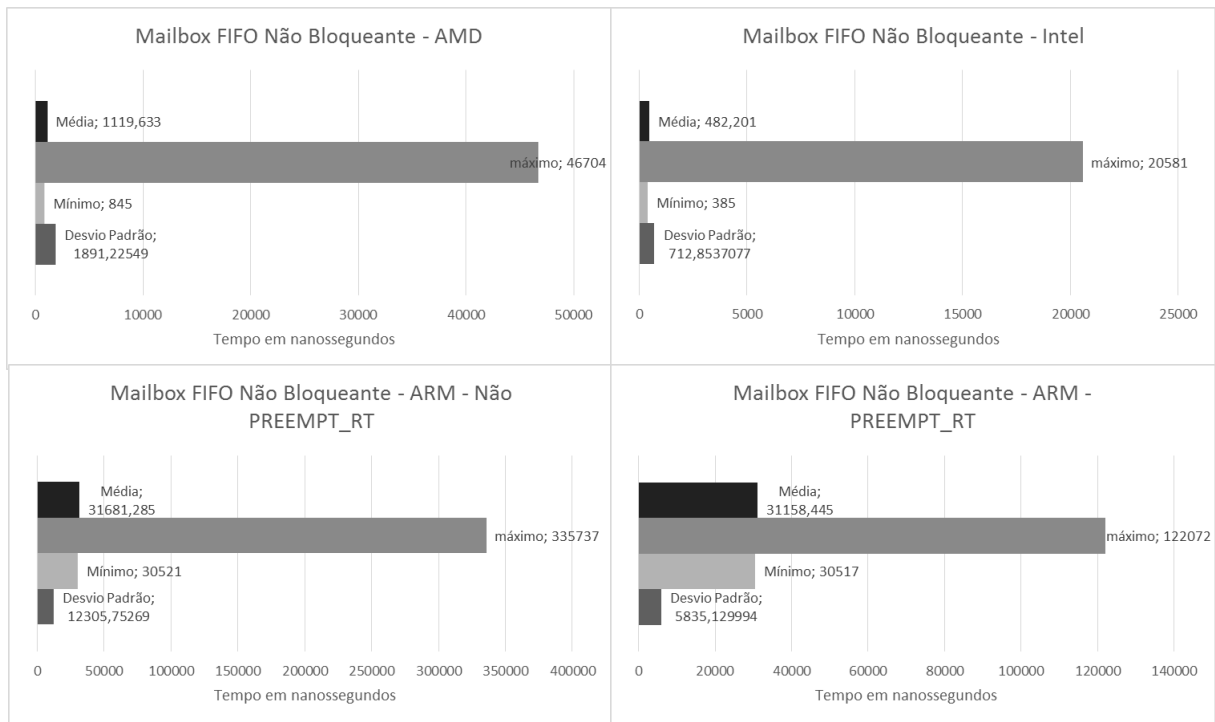


Figura 6.7 - Resultados para o benchmark de mailbox não bloqueante.

Tabela 6.6 - Coeficiente de variação para o benchmark de mailbox não bloqueante.

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREEMPT_RT
168,91	147,83	38,84	18,73

Para o benchmark de mensagens não bloqueantes o pseudocódigo está descrito na *Figura 6.7*, os gráficos com os valores estão na *Figura 6.8* e o coeficiente de variação obtido conforme *Tabela 6.6* para ARM PREEMPT\_RT, ARM, AMD e Intel foi respectivamente: 18,73%, 38,84%, 168,91% e 147,83%. Tendo neste caso a ARM PREEMPT\_RT uma distribuição homogênea.

#### 6.1.4 Fila (*Queue*)

Como meio de comunicação entre as tarefas, a fila preserva a ordem que as mensagens são enviadas: primeira mensagem a ser enviada, primeira mensagem a ser resgatada. Para o kernel RTXC, o conteúdo, ou seja, seu valor é copiado para o *buffer*. Para esse teste foi copiado 1 byte de mensagem para cada iteração, de acordo com o pseudocódigo da *Figura 6.9* e os resultados na *Figura 6.10*, já as demais variações como testes não bloqueante e bloqueantes por tempo determinado encontram-se nos apêndices.

```
void Tarefa1()
{
    Insere um período e torna periódico;

    for(i=0;i<1000;i++)
    {
        Inicia medida de tempo;
        Coloca dado na fila;
        Retira dado da fila;
        Finaliza a medida de tempo;
    }
}
```

**Figura 6.9 - Pseudocódigo para o benchmark da fila bloqueante.**

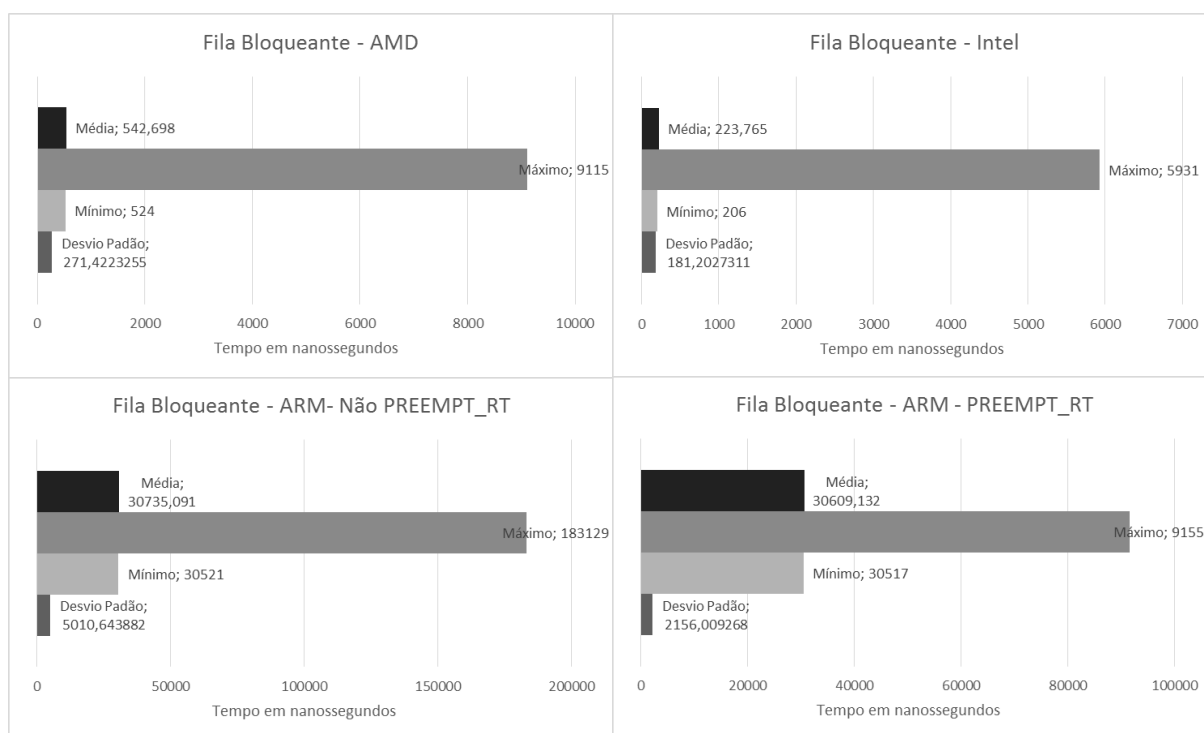


Figura 6.10 - Resultados para o benchmark da fila.

Tabela 6.7 - Coeficiente de variação para o benchmark da fila.

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREEMPT_RT
50,01	80,98	16,30	7,04

Para os valores de desvio padrão sobre média para ARM PREEMPT\_RT, AMD e Intel tem-se: 7,04% (distribuição homogênea), 16,30% (distribuição homogênea), 50,01% e 80,98% respectivamente, conforme Tabela 6.7.

### 6.1.5 Alarme

Alarme tem como função sincronizar a ocorrência de eventos, disparo de funções e ajustes finos de escalonamento. Nesse benchmark é calculado o tempo de disparo de alarme, sua realização envolve diversos objetos do *kernel* como *Event Source*, contadores e alarme. Na implementação interna desse benchmark da API-RTXC-Linux foi definida a resolução interna de 10 nanossegundos esse

valor, entretanto, pode ser alterado de acordo com a necessidade. Vale ressaltar que quanto menor o tempo, mais frequentes são as chamadas internas de verificação do alarme, tendo consequência maior recursos de CPU gastos nessa verificação de escalonamento de alarme. Na *Figura 6.11* temos o pseudocódigo para o benchmark do alarme e na *Figura 6.12* os gráficos com as medidas obtidas.

```
void main()
{
    Define função a ser disparada no alarme;
    Inicia atributos alarme para disparo a cada evento;
    Arma e Inicia alarme;
    Inicia Tarefa1;
}
void Tarefa1()
{
    Inicia a contagem do tempo;
    Processa a fonte de evento;
}
void funcao_disparada_no_alarme(void *arg, void *envarg)
{
    Finaliza a contagem do tempo;
}
```

**Figura 6.11 - Pseudocódigo para benchmark do alarme.**

**Tabela 6.8 - Coeficiente de variação para benchmark do alarme.**

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREMPT_RT
24,73	6,52	29,13	17,22

Para coeficiente de variação os valores obtidos foram: 17,22%(distribuição homogênea), 29,13%, 24,73% e 6,52%(distribuição homogênea) para ARM PREEMPT\_RT, ARM, AMD e Intel de acordo com *Tabela 6.8*.

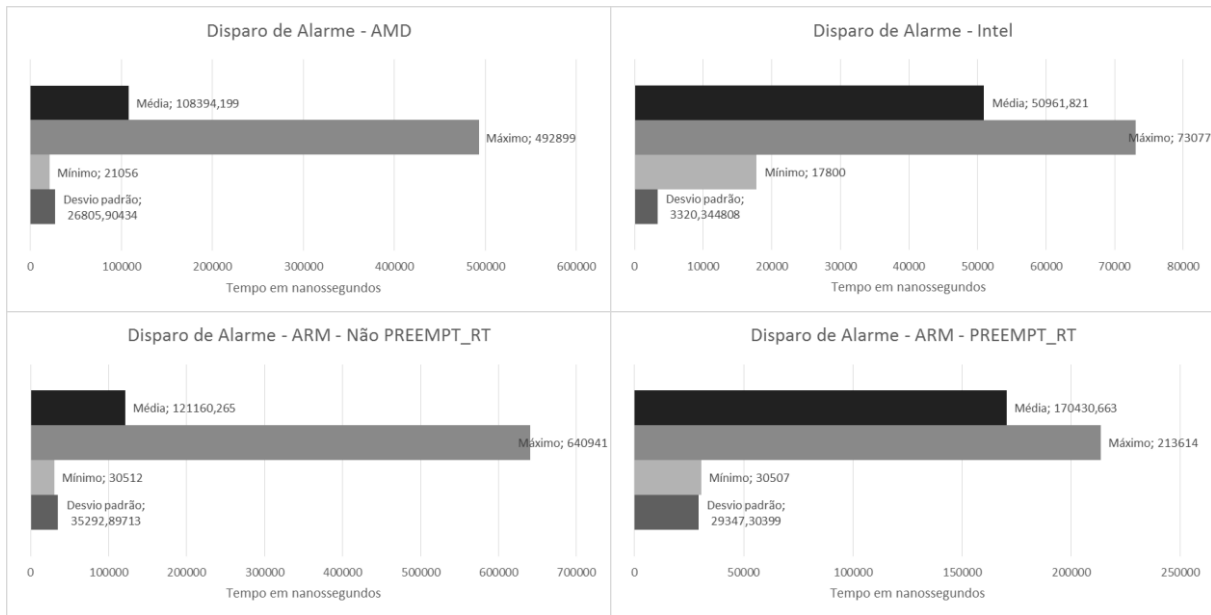


Figura 6.12 - Gráficos do benchmark para disparo de alarme.

### 6.1.6 Mutex

Mutex é o objeto responsável pelo gerenciamento de recursos compartilhados do sistema. Sua utilização permite o acesso atômico a variáveis ou recursos, garantindo a coerência em instâncias que executem trabalho concorrentemente. Na *Figura 6.13*, tem-se o pseudocódigo dos testes de desempenho realizados.

```

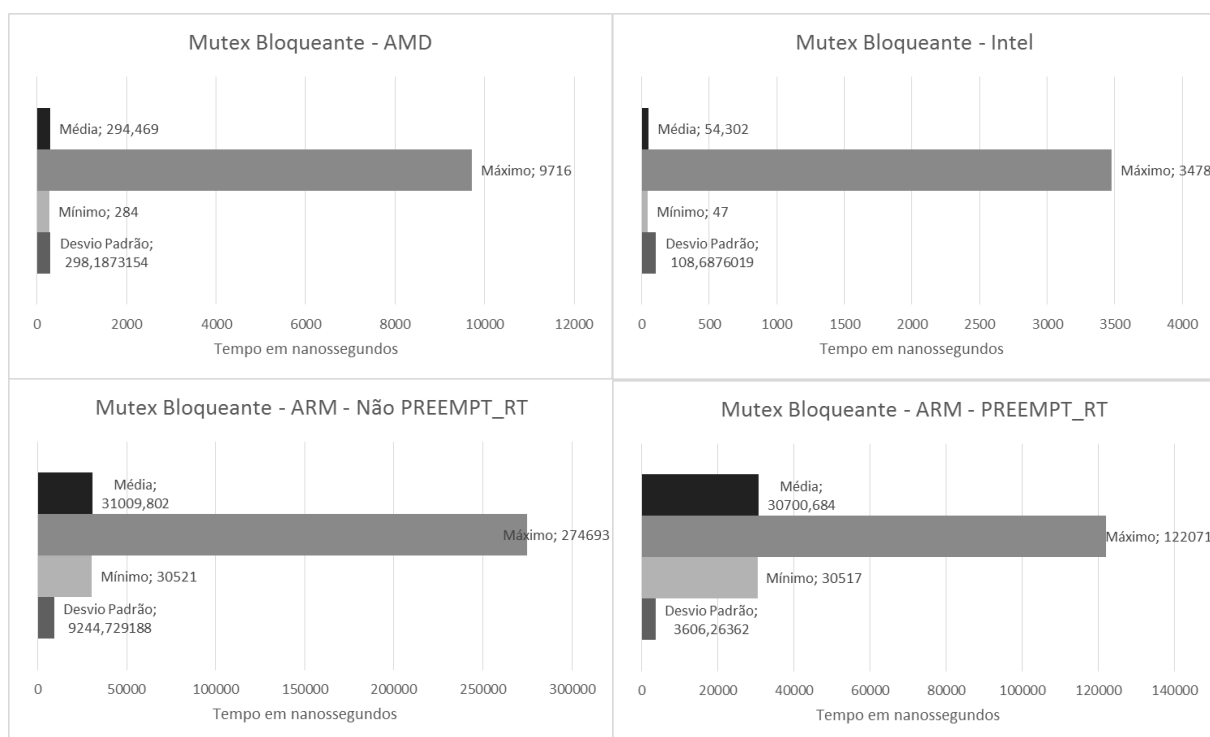
void Tarefa1()
{
    Insere um período e torna periódico;

    for(i=0;i<1000;i++)
    {
        Inicia medida de tempo;
        Obtém o Mutex;
        Libera o Mutex;
        Finaliza a medida de tempo;
    }
}

```

Figura 6.13 - Pseudocódigo para o benchmark do mutex.

Na *Figura 6.14* tem-se os gráficos com os tempos do benchmark sendo que as demais variações do teste como não bloqueante e bloqueante por tempo determinado encontram-se nos apêndices. O coeficiente de variação é: 11,75%(distribuição homogênea), 29,81%, 101,26% e 200,15% para ARM PREEMPT\_RT, ARM, AMD e Intel respectivamente conforme *Tabela 6.9*.



**Figura 6.14 - Gráficos do benchmark para mutex bloqueante.**

**Tabela 6.9 - Coeficiente de variação para mutex bloqueante.**

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREEMPT_RT
101,26	200,15	29,81	11,75

### 6.1.7 Semáforo

O semáforo é responsável por controlar o acesso a um recurso compartilhado por várias tarefas. A versão bloqueante do semáforo, no caso em que não exista mais recurso faz com que a tarefa espere até que esse recurso seja liberado para uso. Na *Figura 6.15* temos um exemplo do teste efetuado para o semáforo.

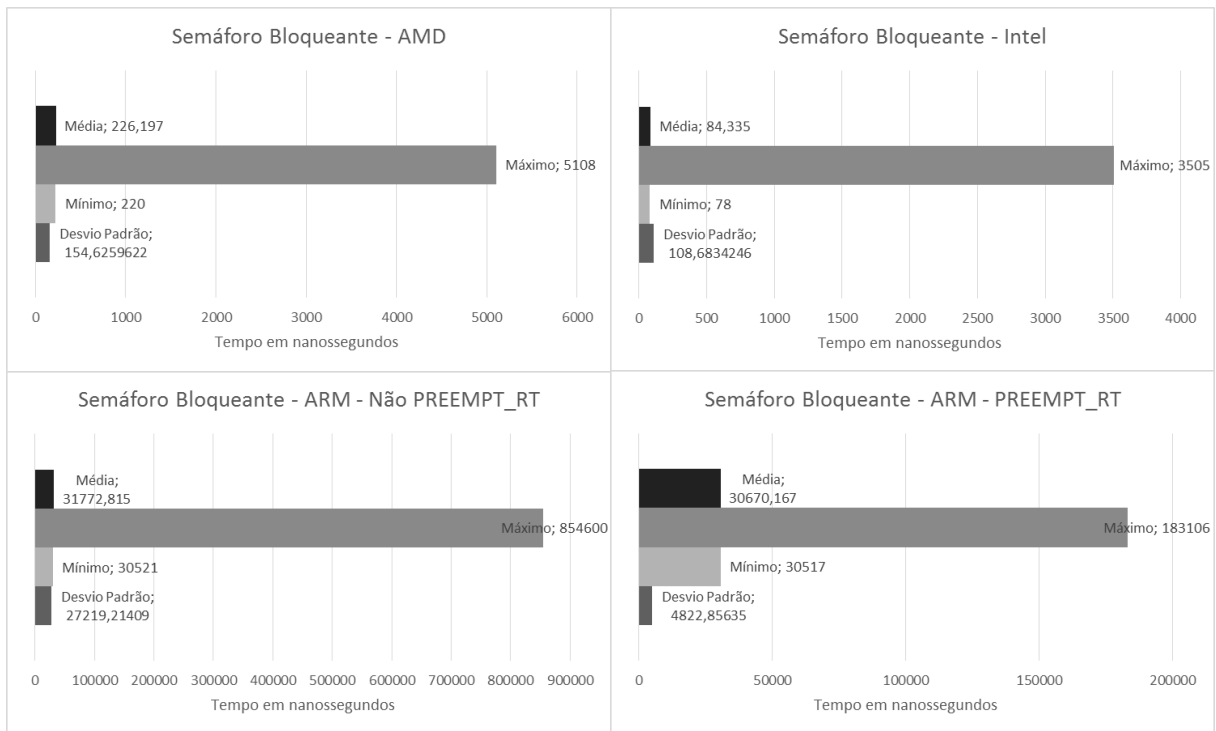
```

void Tarefa1()
{
    Insere um período e torna periódico;

    for(i=0;i<1000;i++)
    {
        Inicia medida de tempo;
        Obtém semáforo;
        Libera semáforo;
        Finaliza a medida de tempo;
    }
}
    
```

**Figura 6.15 - Pseudocódigo para o benchmark de semáforo.**

Na *Figura 6.16* temos os resultados do benchmark para o semáforo bloqueante, as demais variantes encontram-se nos apêndices. Como coeficiente de variação tem-se: 15,72%(distribuição homogênea), 85,87%, 68,36% e 128,87% para ARM PREEMPT\_RT, ARM, AMD e Intel respectivamente conforme *Tabela 6.10*.



**Figura 6.16 - Resultados para o benchmark de semáforo.**

**Tabela 6.10 - Coeficiente de variação para o benchmark de semáforo.**

Coeficiente de variação (%)			
AMD	Intel	ARM	ARM PREEMPT_RT
68,36	128,87	85,67	15,72



### 6.1.8 Resumo dos resultados benchmark

Como relação aos resultados obtidos, incluindo aqueles que os gráficos estão nos apêndices na *Tabela 6.11* é sintetizado o coeficiente de variação de cada uma das arquiteturas utilizadas.

**Tabela 6.11 - Coeficiente de Variação das medidas do Benchmark**

Benchmark	Coeficiente de Variação (%)			
	AMD	Intel	ARM - Não PREEMPT_RT	ARM PREEMPT_RT
Alarme	24,73	6,52	29,13	17,22
Troca de Contexto	111,97	66,35	73,11	22,35
Mailbox Bloqueante	63,23	57,05	46,40	13,32
Mailbox Bloqueante por Tempo Determinando	43,45	49,93	39,55	19,17
Mailbox Não Bloqueante	168,91	147,83	38,84	18,73
Mutex Bloqueante	101,26	200,15	29,81	11,75
Mutex Bloqueante por Tempo Determinado	69,99	111,42	22,56	7,04
Mutex Não Bloqueante	106,83	160,26	22,99	4,46
Partição Requisição Bloqueante	78,79	123,54	28,55	14,65
Partição Requisição Bloqueante por Tempo Determinado	184,41	114,06	29,29	5,45
Partição Requisição Não Bloqueante	79,65	121,69	24,00	8,87
Fila Bloqueante	50,01	80,98	16,30	7,04
Fila Bloqueante por Tempo Determinado	108,54	51,25	25,36	10,39
Fila Não Bloqueante	32,85	60,75	25,80	7,70
Semáforo Bloqueante	68,36	128,87	85,67	15,72
Semáforo Bloqueante por Tempo Determinado	44,64	52,39	22,12	11,30
Semáforo Não Bloqueante	62,82	61,47	22,26	10,87

Observa-se que para ARM PREEMPT\_RT praticamente todos os coeficientes estão dentro do intervalo de uma distribuição homogênea ou bem próximos desse critério, indicando a tendência dos tempos de resposta serem determinísticos.

## 6.2 Testando a API com Mandelbrot Multicore

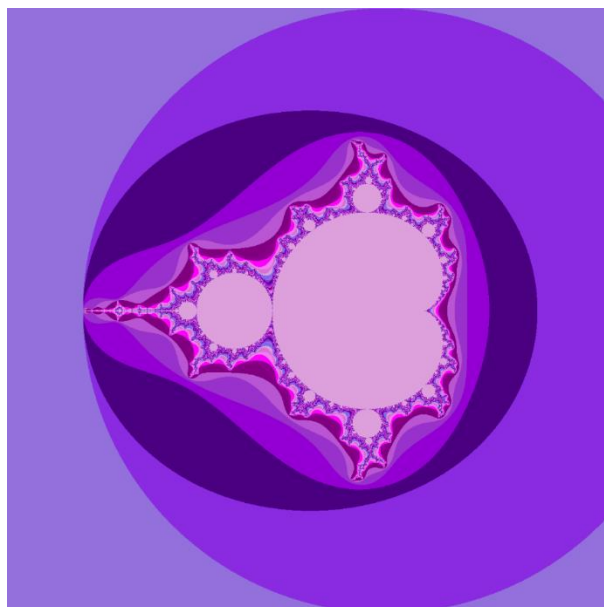
Para teste do ambiente multicore, além do benchmark realizado com o objeto mailbox usando dois cores, foi realizado o teste da API com o Mandelbrot Multicore.

O conjunto de Mandelbrot é um arranjo de números complexos obtidos através da fórmula de recorrência:

$$Z_{n+1} = Z_n^2 + c$$

Onde  $Z_0 = C$ , e  $C$  são pontos do plano complexo.

O teste foi escrito na linguagem C usando a API em duas versões: uma utilizando uma Tarefa, e outra usando duas Tarefas associadas a dois núcleos. O número máximo de iterações definidas foi 1000, e intervalo do eixo calculado foi para o Eixo X= [-2,5; 1,5] e para o Eixo Y = [-2,0;2,0]. O tamanho da imagem calculada foi de 1280x1280 pixels foi utilizada uma paleta de 13 cores de acordo com o número iterações em cada pixel, obtendo-se a imagem da *Figura 6.17*. O teste *multitask* foi realizado usando dois cores, pois o processador ARM possui apenas dois cores e o objetivo era compará-la com AMD e Intel.



**Figura 6.17 - Mandelbrot obtido no benchmark.**

Para os benchmarks realizados os tempos obtidos estão na *Tabela 6.12*. O ganho do teste realizado usando duas Tarefas e dois núcleos em relação ao de um núcleo e uma Tarefa foi: 99,71%, 96,24%, 63,67% e 67,34% para AMD, Intel, ARM e ARM PREEMPT\_RT respectivamente conforme *Tabela 6.13*.

Tabela 6.12 - Tempo para o teste Mandelbrot

Benchmark	Tempo em segundos			
	AMD	Intel	ARM - Não PREEMPT_RT	ARM PREEMPT_RT
1 Tarefa, 1 core	3,332	1,164	10,565	11,504
2 Tarefa, 2 cores	1,668	0,593	6,451	6,874

Tabela 6.13 - Ganho no teste Mandelbrot com duas Tarefas e dois cores.

Ganho (%)			
AMD	Intel	ARM - Não PREEMPT_RT	ARM PREEMPT_RT
99,71	96,24	63,76	67,34

Esses valores mostram que a API tem um bom desempenho para aplicações *multitask* e *multithreading*. Os tempos e ganhos para o aumento de Tarefas foram diferentes para cada um dos elementos testados, o que se deve ao fato das características específicas de cada arquitetura, além de diferenças do *kernel* no caso de ARM que são o mesmo processador. Mesmo sendo o kernel PREEMPT\_RT não voltado para se obter a melhor vazão, mas sim para obter um bom tempo de resposta o ganho para duas tarefas foi maior em porcentagem que no kernel convencional, entretanto os valores absolutos de tempo no *kernel* PREEMPT\_RT foram ligeiramente maiores.

# Capítulo 7

## CONCLUSÃO

---

O advento dos processadores multicore possibilitou que o aumento do desempenho dos sistemas computacionais se direcionasse para replicação de núcleos ao invés do aumento da frequência do *clock*. Essa replicação ocorreu graças ao desenvolvimento tecnológico que permitiu que se encapsulassem dois ou mais núcleos em um mesmo processador. Além das melhorias no processador, o surgimento de barramentos mais velozes e novas arquiteturas permitiram a aglomeração de vários processadores em uma mesma máquina.

Os sistemas embarcados seguiram essa tendência ao utilizarem processadores *multicore*, melhorando o poder computacional disponível para sistemas de tempo real. A melhoria adicionada ao hardware dos sistemas embarcados abriu espaço para que a cadeia de desenvolvimento GNU/Linux pudesse ser executada nessas plataformas. O Linux não é um sistema operacional de tempo real, sua execução busca realizar o melhor uso da CPU de forma que a melhor vazão seja atingida ao invés de tornar determinístico o processamento das saídas.

Dada a complexidade dos projetos atuais para aplicações críticas, tem sido necessário o uso de *kernel* de tempo real nesses desenvolvimentos. Um dos *kernels* mais tradicionais é o RTXC, que foi desenvolvido atendendo os requisitos de aplicações dessa área. Nossa proposta é trazer para o Linux toda experiência adquirida com o *kernel* de tempo real, através de uma biblioteca que emule o *kernel* de tempo real RTXC.

Três principais abordagens buscam trazer o sistema operacional Linux para o contexto de tempo real sendo: a primeira, a criação de um *kernel* extra para atender os requisitos de tempo real; a segunda, a estratégia de se evitar utilizar o espaço do

*kernel* usando o espaço de usuário e a terceira, sendo uma abordagem mais extrema que é a modificação de diversas áreas do *kernel* para adequar ao escalonamento de tempo real. Usando a terceira opção, juntamente com a biblioteca Pthreads foi implementada uma biblioteca que permite que os mesmos programas que executam usando RTXC, executem na nossa API-RTXC-Linux.

A API foi construída usando a linguagem C e o padrão POSIX para threads ou Pthreads sendo, portanto, possível compilá-la nativamente em diversas arquiteturas. Cada um dos três níveis foi implementado: Zona 1 - IS, nível de interrupção; Zona 2 - TS, nível de Thread e Zona 3 – KS, nível de Tarefa. A existência de manual detalhado para o kernel RTXC permitiu a emulação de funções e seus comportamentos mantendo a correspondência da assinatura de cada uma das chamadas.

## 7.1 Uso da API-Linux-RTXC

Para o contexto da educação o uso da API pode ser um grande aliado possibilitando o aprendizado de programação paralela em tempo real. Isso é facilitado pelo fato de sua base ser Linux, portanto pode-se executá-la em qualquer computador pessoal dispensando a necessidade de um hardware dedicado, ao mesmo tempo que permite essa transição de maneira fácil para todos os embarcados compatíveis com o Linux. Sua utilização não requer configurações complexas, bastando apenas que se recompile para a plataforma adotada.

Outra possibilidade de uso é a migração de projetos feitos em outras plataformas ou sistemas operacionais para o ambiente Linux. Isso permite a transição de ambientes proprietários para ambientes livres baseados em Linux, reduzindo custos com licenças.

Devido aos conceitos e recursos fornecidos na API que são característicos de ambientes complexos de tempo real, a comunidade Linux ganhará novas possibilidades com o uso da API.

## 7.2 Contribuições e Limitações

Diversos testes foram realizados na API desenvolvida e pôde-se observar resultados muito bons para API-RTXC-Linux + kernel Linux PREEMPT\_RT, obtendo distribuição homogênea nos tempos de resposta em 94,4% dos testes realizados. Esses resultados mostraram tendência a tempos de resposta determinísticos, característica essencial para um sistema de tempo real apresentando, portanto, pouca variação entre as amostras. A API-RTXC + PREEMPT\_RT é adequada para aplicações de tempo real desde que estejam dentro do escopo de tempo de respostas obtidos para cada objeto.

Pretende-se distribuir a API desenvolvida sob a licença GNU GLP, permitindo o acesso ao código fonte e que se agreguem melhorias e se distribua livremente para comunidade. O acesso ao código fonte favorece não apenas melhorias, mas um avanço no aprendizado de desenvolvimento de aplicações paralelas e requisitos de sistemas de tempo real, podendo estudar as principais estratégias adotadas na solução dos desafios da programação da API. Além disso, a implementação dos objetos do kernel podem ser modificadas para atender certas especificidades.

Com o uso da API-Linux-RTXC vai ser possível uma troca de módulos implementados de tal forma a criar uma base de onde é possível criar novos sistemas. Essa estratégia é impossível de ser implementada utilizando-se software proprietário.

A maior limitação é que a API terá um tempo maior de execução que uma distribuição do RTOS RTXC *baremetal*, isso se deve ao fato de existirem mais camadas. Portanto, quando se trabalhar com restrições temporais menores que os resultados apresentados, a API não poderá ser usada. Outra limitação é que o hardware precisa ser compatível com o ambiente GNU/Linux, em hardwares mais simples a API Linux RTXC poderá não compilar.

### 7.3 Trabalhos Futuros

O passo seguinte para termos um desenvolvimento de sistemas mais robusto é a definição de uma metodologia para o desenvolvimento de sistemas multicore baseado no nosso modelo multitask. A metodologia tradicional de desenvolvimento envolve quatro etapas: análise, design, implementação e inspeção. A nova metodologia para multicore necessita acrescentar mais duas etapas que são: mapeamento de tarefas em núcleos e a otimização da aplicação (MORÓN et al., 2014).

Outro trabalho a ser feito é o estudo da usabilidade da integração com o ambiente visual. Neste ambiente, os componentes de objetos do *kernel* são diagramados e interagem entre si, tornando seus relacionamentos mais claros. Isso permite ao programador uma visão geral dos componentes importantes do *kernel* e visualizar rotinas paralelas não tão perceptíveis na linha de código.

# REFERÊNCIAS

---

Allinea DDT. Disponível em: <<https://computing.llnl.gov/?set=code&page=ddt>>. Acesso em 07 mar. 2016.

AMDAHL, G. M. Validity of the single-processor approach to achieving large scale computing capabilities. **AFIPS Conference Proceedings**, v. 30, p. 483–485, 1967.

AUSTIN, T.; ARBOR, A.; LARSON, E.; ERNST, D.; Michigan Univ. SimpleScalar: an infrastructure for computer system modeling. **IEEE Computer Society**, v.35, issue: 2, p. 59 – 67, 2002.

CFS Scheduler. Disponível em: <<https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>>. Acesso 24 mar.2016.

CHOUGULE M.D.; GUTTE P. H. Parallel Programming Models: A Systematic Survey. **(IJCSIT) International Journal of Computer Science and Information Technologies**, Vol. 5 (4), 5268-5271, 2014.

Cortex-A72 Processor – ARM. Disponível em: <<https://www.arm.com/products/processors/cortex-a/cortex-a72-processor.php>>. Acesso 26 mar.2016.

BARNEY, B. Introduction to Parallel Computing. Livermore Computing, 2015. Disponível em:<[https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)>. Acesso 28 fev.2016.

BARNEY, B. TotalView. Livermore Computing, 2015. Disponível em:<<https://computing.llnl.gov/tutorials/totalview/>>. Acesso 28 fev.2016.

BERGER, A. S. **Embedded Systems Design: An Introduction to Processes, Tools, and Techniques**.1ª ed. CMP Books, Taylor & Francis, 2002.

BLAIR, S.; STOKES, A. **Parallel Programming with Intel Parallel Studio XE**. Wrox, 2012.

BULL J. M.; ENRIGHT J.; GUO X.; MAYNARD C.; REID F. Performance Evaluation of Mixed-Mode OpenMP/MPI Implementations, **International Journal of Parallel Programming**, Vol.38, Issue 5, p. 396-417, 2010.

FLYNN, M. J. Some Computer Organizations and Their Effectiveness. **IEEE Transactions on Computers**, v. C-21, n. 9, p. 948–960, 1972.

Intel® Parallel Studio XE 2016. Disponível em:<<https://software.intel.com/en-us/intel-parallel-studio-xe/details>>. Acesso 03 mar.2016.



Inside the Linux 2.6 Completely Fair Scheduler. Disponível em: <<http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>>. Acesso 03 jan.2017.

IVICA, C.; RILEY, J. T.; SHUBERT, C. StarHPC - Teaching Parallel Programming within Elastic Compute Cloud. **ITI 2009 31st International Conference on Information Technology Interfaces**, p. 353 – 356, 2009.

KAHLE, J. A.; DAY, M. N.; HOFSTEE, H. P.; MAEURER, T. R.; SHIPPY, D. Introduction to the Cell multiprocessor. **IBM Journal of Research and Development**, vol. 49, Issue 4/5, Pag.589-604, 2005.

KOUFATY, D.; MARR, D. T. Hyperthreading Technology In The Netburst Microarchitecture. **IEEE Micro**, Vol. 23, Issue 2, p. 56-65, 2003.

LABROSSE, J.; GANSSE, J.; NOERGAARD, T.; OSHANA, R.; WALLS, C.; CURTIS, K. ; ANDREWS, J.; KATZ, D. J.; GENTILE, R.; HYDER, K.; PERRIN, B. **Embedded Software**. 1ª ed. Elsevier, 2008.

LIN C.; SNYDER L. **Principles of Parallel Programing**. 1ª ed. Addison-Wesley, 2009.

Linux kernel configuration for 2.6.18 on i386. Disponível em: <<http://kernel.xc.net/html/linux-2.6.18/i386/>>. Acesso em 14 mar. 2016.

LIPARI, G. Real-Time Linux and the Xenomai system, 2008. Disponível em: <<http://retis.sssup.it/~lipari/courses/str07/xenomai.pdf>>. Acesso em: 17 mar. 16.

MOLKA, D.; HACKENBERG, D.; SCHÖNE, R. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. **MSPC '14 Proceedings of the workshop on Memory Systems Performance and Correctness**, n. 4, 2014.

MORÓN, C. E.; IDEGUCHI, A.; FERNANDES M. M. From MultiTask to MultiCore: Design and Implementation Using na RTOS. **IEEE 13th International Symposium on Parallel and Distributed Computing (ISPDC)**, pag. 111 – 118, 2014.

PFEIFFER, W.; STAMATAKIS, A. Hybrid MPI/Pthreads Parallelization of the RAXML Phylogenetics Code. **IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)**, pag. 1 - 8, 2010.

PERISSATTO, M. G.; GONÇALVES JÚNIOR, N. A.; GONÇALVES, R. A. L.; MARTINI, J. A. Ferramenta para Simulação de Multiprocessadores Superescalares de Memória Compartilhada. **Workshop sobre Educação em Arquitetura de Computadores (WEAC)**, 2007.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. 1ª ed. McGrawHill Higher Education, 2004.

Rauber, G.; Raugber T. **Parallel Programming for Multicore and Cluster Systems**. 1<sup>st</sup> Ed. Springer-Verlag Berlin Heidelberg, 2012.

RTXC Quadros Manuals. Disponível em: <[http://www.nevis.columbia.edu/~chi/NCC/n3c/quadros/Help/pdf/rtxc\\_users\\_guidev1.pdf](http://www.nevis.columbia.edu/~chi/NCC/n3c/quadros/Help/pdf/rtxc_users_guidev1.pdf)>

RTXC Quadros Real-Time Operating System. Disponível em: <<http://quadros.com/products/operating-systems/rtxc-quadros-rtos-advanced/>>.

Acesso em: 17 mar. 16

Start Here – Xenomai. Disponível em: <<http://quadros.com/products/operating-systems/rtxc-quadros-rtos-advanced/>>. Acesso em: 10 mar. 16

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating system concepts**. 9<sup>o</sup> ed. John Wiley & Sons, Inc., 2013.

T.C, D. S.; VARAGANTI, K.; SURESH, R.; GARG, R.; RAMAMOORTHY, R. Comparison of Parallel Programming Models for Multicore Architectures. **IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)**, p. 1675 - 1682, 2011).

Wael, M. DE; MARR, S.; FRAINE, B. DE; et al. Partitioned Global Address Space Languages. **Journal ACM Computing Surveys (CSUR)**, V. 47 Issue 4, n. 62, 2015.

Yang, C.; Chih-Lin Huang, Cheng-Fang Lin - Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. **Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung**, Pag. 266–269, 2009.

# Apêndice A

## GRÁFICOS DO BENCHMARK

Nessa seção serão colocados os gráficos que não foram apresentados no Capítulo 6.

### 1. Benchmark Partição Não Bloqueante/Bloqueante por tempo determinado

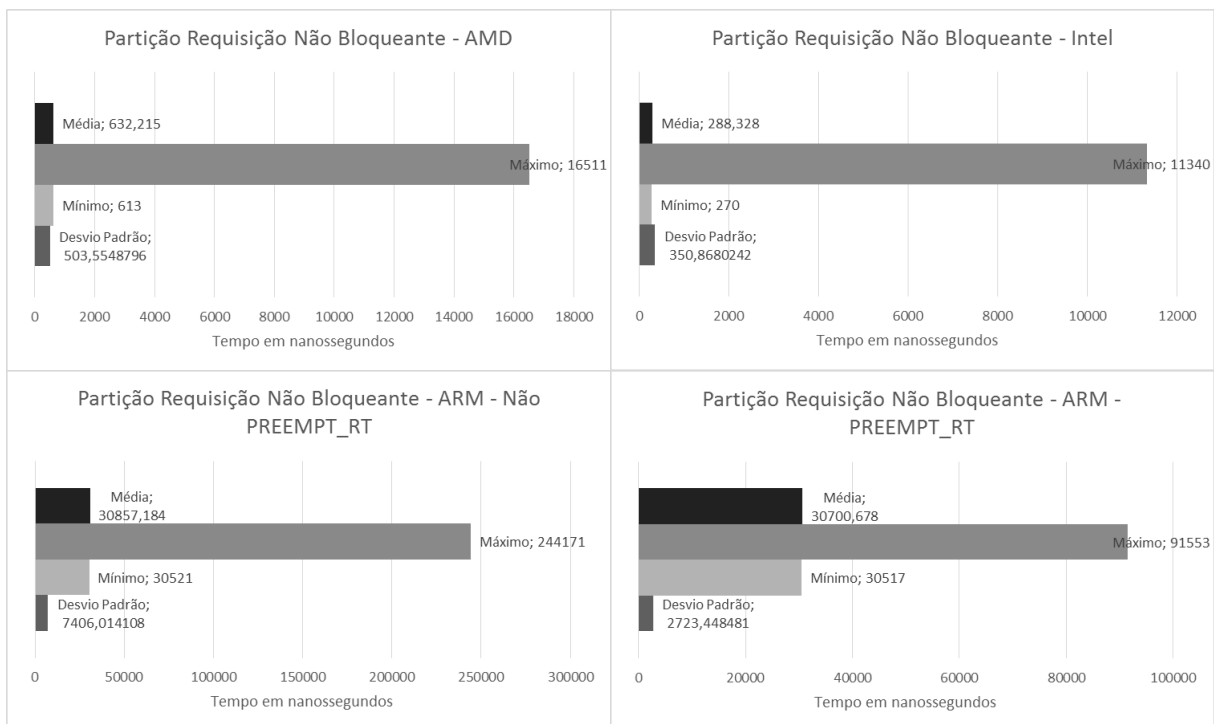
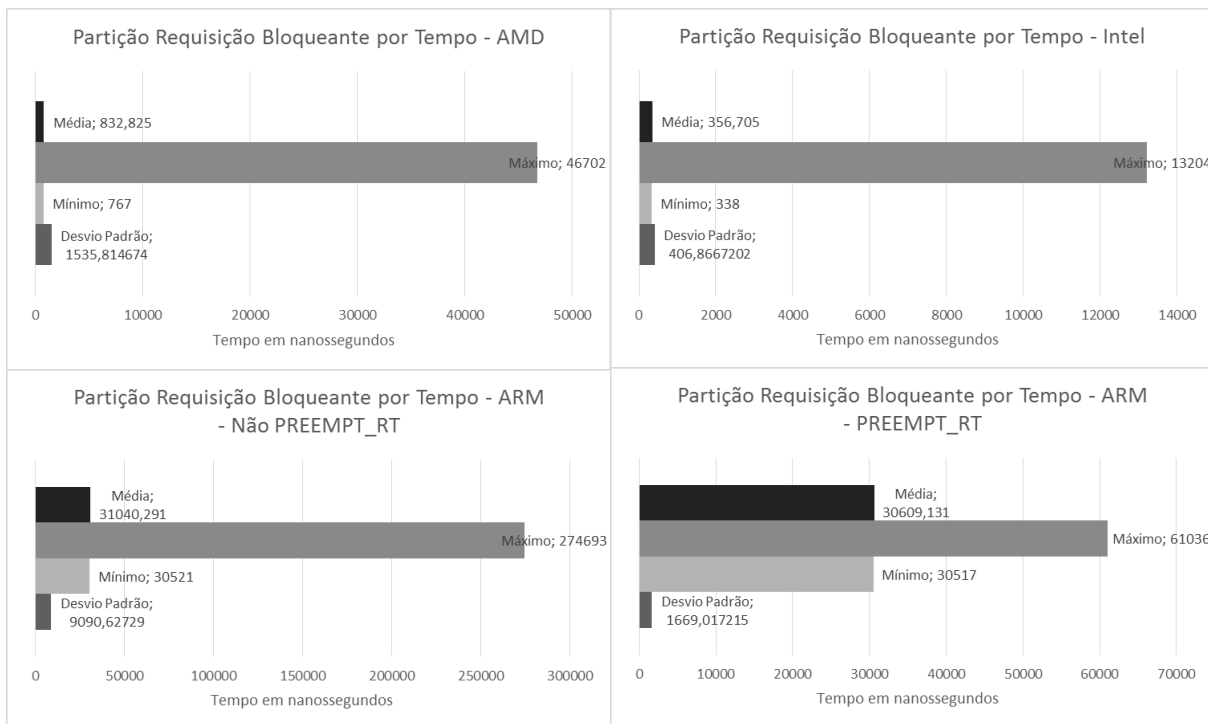
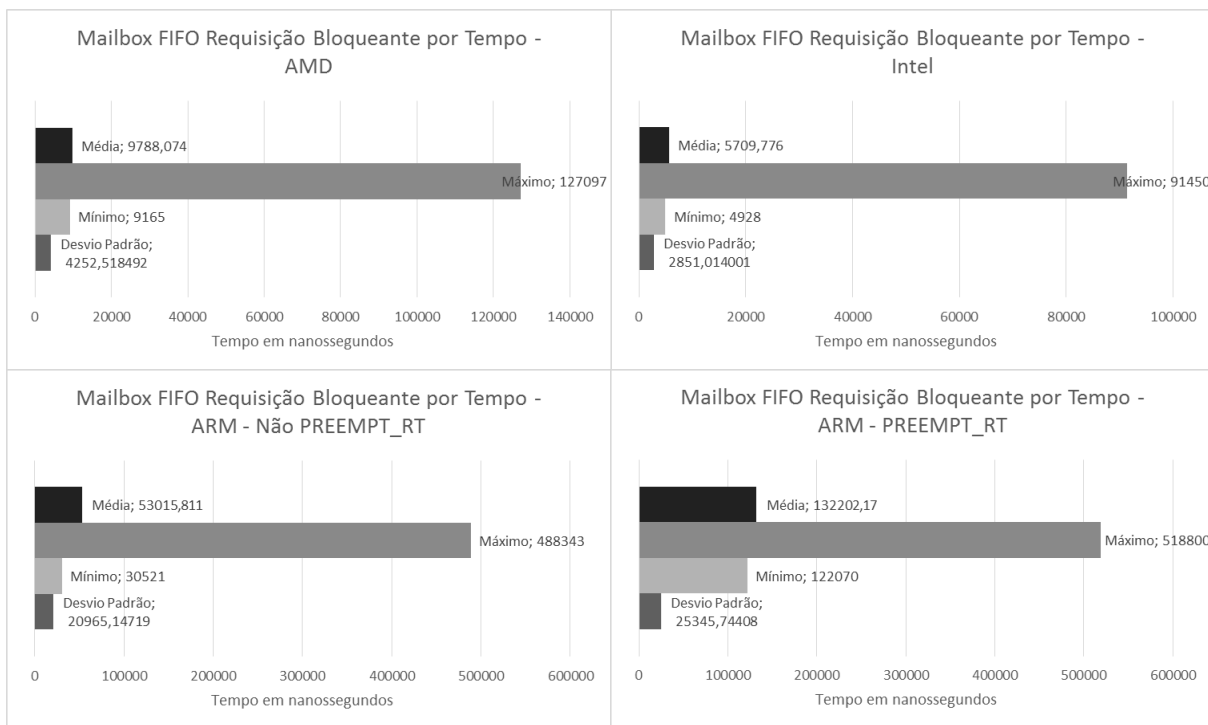


Figura Apêndice 1 - Resultados Benchmark Partição Requisição não-bloqueante



**Figura Apêndice 2 - Resultados Benchmark Partição Requisição Bloqueante por Tempo Determinado.**

## 2. Benchmark Mailbox Requisição Bloqueante por Tempo Determinado



**Figura Apêndice 3 – Resultados Benchmark Mailbox Requisição Bloqueante por Tempo Determinado.**

### 3. Benchmark Fila Não-Bloqueante/Bloqueante por Tempo Determinado

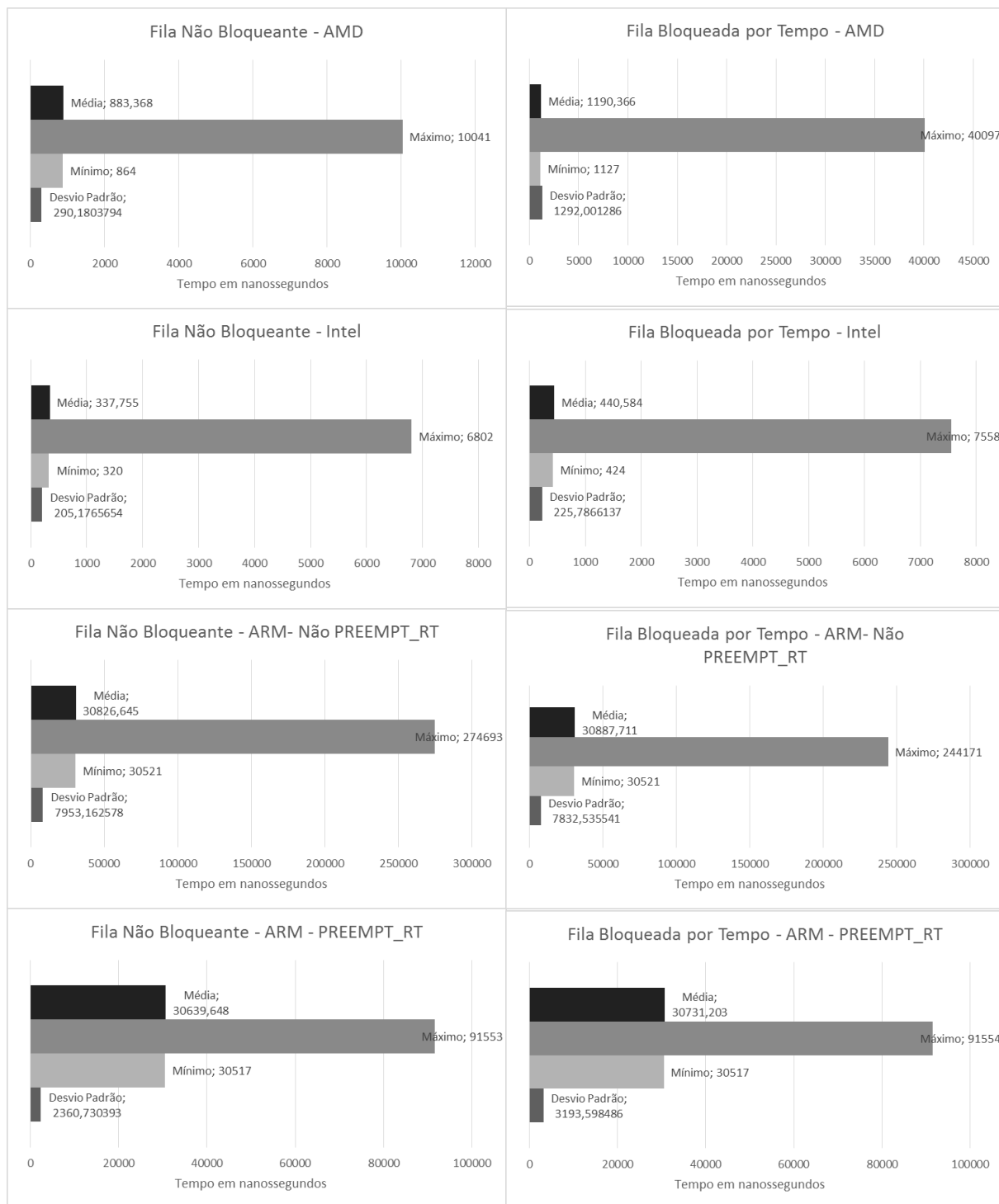


Figura Apêndice 4 – Resultados Benchmark Fila Não Bloqueante/Bloqueante por Tempo.

### 4. Benchmark Mutex Não-Bloqueante/Bloqueante por Tempo Determinado

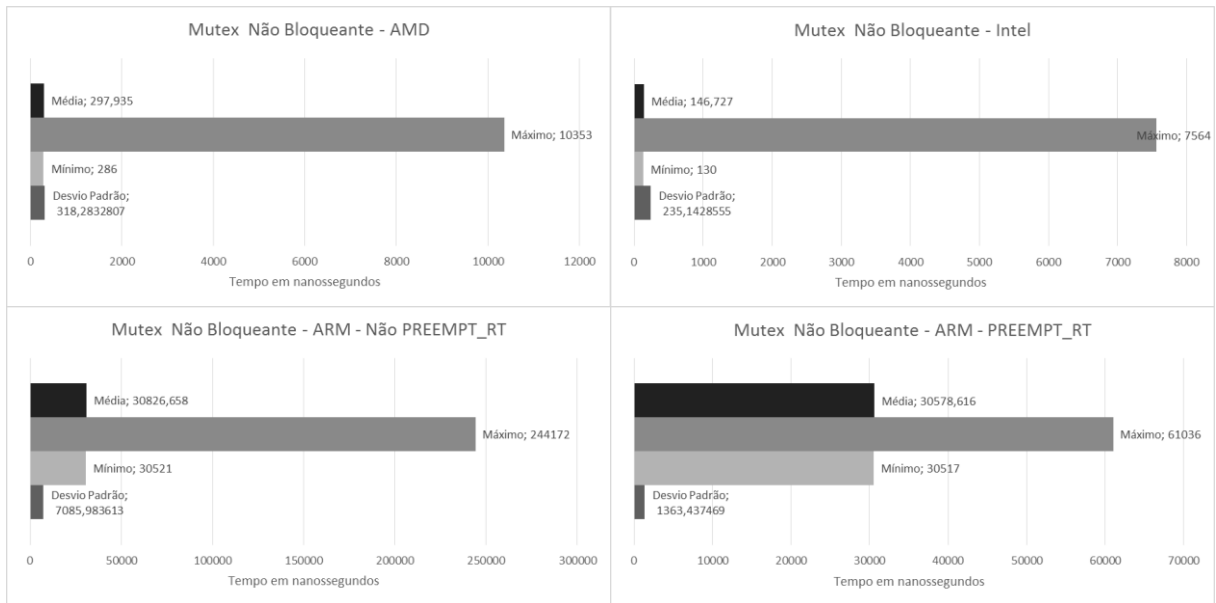


Figura Apêndice 6 - Resultado Benchmark Mutex Não-Bloqueante.

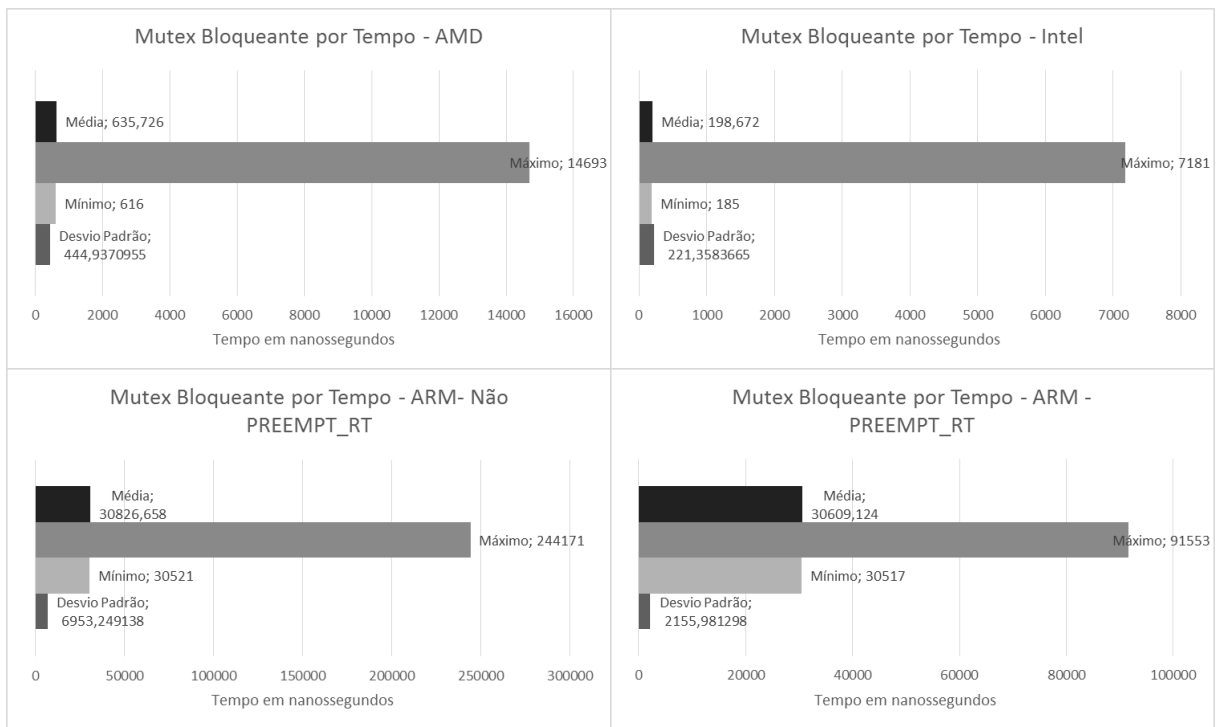


Figura Apêndice 5 - Resultado Benchmark Mutex Bloqueante por Tempo.

### 5. Benchmark Semáforo Não-Blockeante/Blockeante por Tempo Determinado

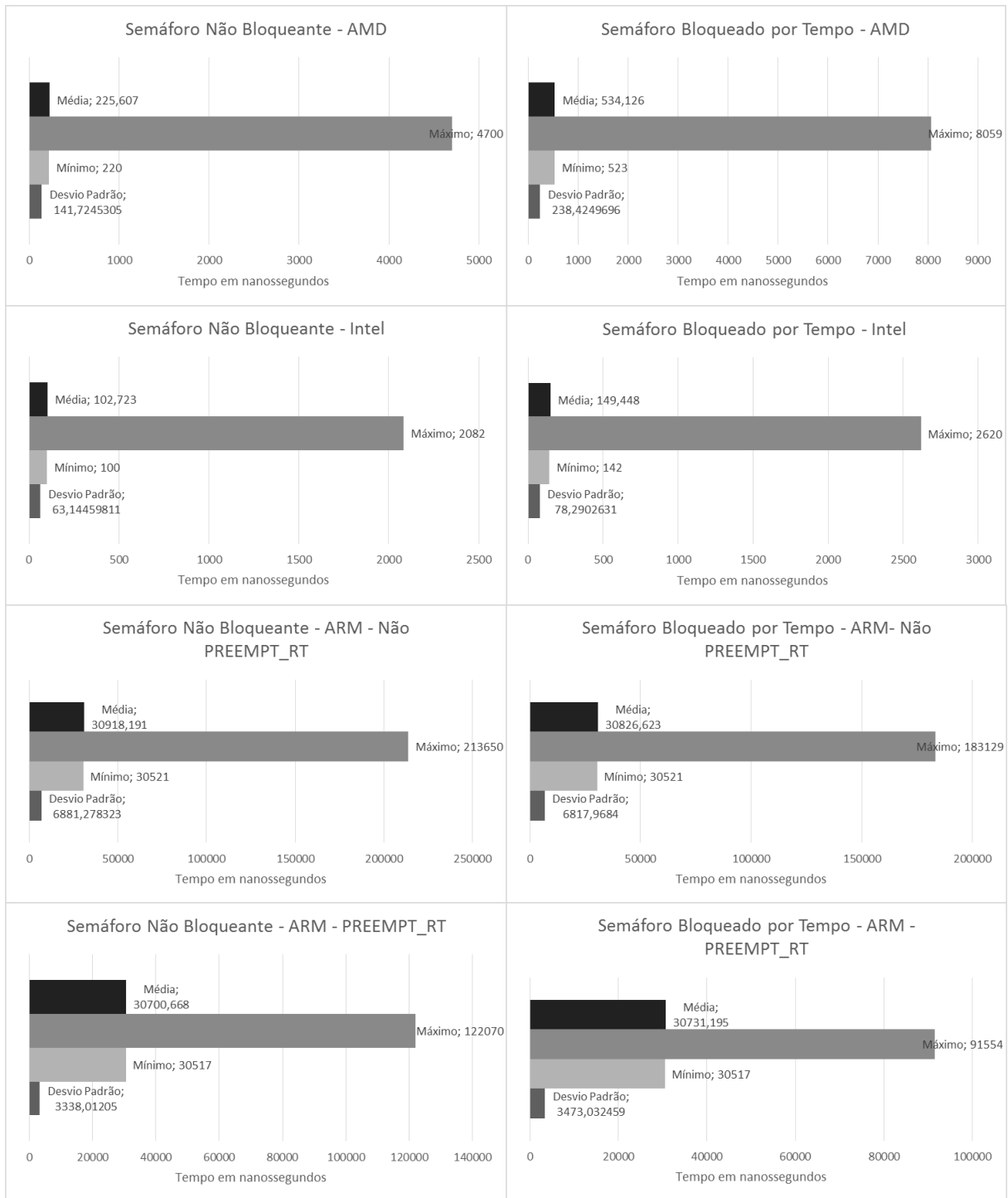


Figura Apêndice 7 - Benchmark Semáforo Não-Blockeante/Blockeante por Tempo Determinado