INSTITUTO FEDERAL DE CIÊNCIA E TECNOLOGIA DE PERNAMBUCO
CAMPUS RECIFE
DEPARTAMENTO ACADÊMICO DE SISTEMAS, PROCESSOS E CONTROLES ELETRO-ELETRÔNICOS
CURSO DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

DANIEL BARLAVENTO GOMES

AVALIAÇÃO DE DESEMPENHO DE SISTEMAS LINUX DE TEMPO REAL: teste comparativo das soluções PREMPT_RT e RTAI

Recife
2018

DANIEL BARLAVENTO GOMES

AVALIAÇÃO DE DESEMPENHO DE SISTEMAS LINUX DE TEMPO REAL: teste comparativo das soluções PREMPT_RT e RTAI

Trabalho de conclusão de curso apresentado ao curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco, como requisito parcial para a obtenção do título de tecnólogo em Tecnologia em Análise e Desenvolvimento de Sistemas.
Orientador: Prof. Mestre Paulo Abadie Guedes

Recife
2018

Ficha elaborada pela bibliotecária Ana Lia Evangelista CRB4/974

G633a

Gomes, Daniel Barlavento.
Avaliação de desempenho de Sistemas Linux de tempo real: teste comparativo das soluções PREMPT_RT e RTAI. / Daniel Barlavento Gomes. Recife, PE: O autor, 2018.
38f. il. color. : 30 cm.

TCC (Tecnólogo em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Pernambuco, DASE 2018.

Inclui Referências e apêndices

Orientador: Professor MsC. Paulo Abadie Guedes.

1. Linux. 2. Tempo real. 3. Sistemas operacionais. 4. Preempt RT. 5. RTAL. I. Guedes, Paulo Abadie (Orientador). II. Título.

CDD 005.43 (21ed.)

Trabalho de Conclusão de Curso apresentado pelo aluno **Daniel Barlavento Gomes** à coordenação de Análise e Desenvolvimento de Sistemas, do Instituto Federal de Pernambuco, sob o título de "***AVALIAÇÃO DE DESEMPENHO DE SISTEMAS LINUX PARA TEMPO REAL Teste comparativo das soluções patch Preempt_RT e RTAI.***", orientado pelo **Prof. Ms. Paulo Abadie Guedes** e aprovado pela banca examinadora formada pelos professores:

Recife, 26 de novembro de 2018.

Prof. Ms. Paulo Abadie Guedes
CTADS/DASE/IFPE

Prof. Dr. Paulo Maurício Gonçalves Júnior
CTADS/DASE/IFPE

Prof. Ms. José Paulo da Silva Lima
Faculdade Nova Roma

Daniel Barlavento Gomes

Recife

2018

AGRADECIMENTOS

Agradeço a meus pais Marcos Antonio da Rocha Gomes e Cristina Maria Barlavento Gomes por sempre terem me dado a liberdade da escolha. A minha avó Vanilda Negromonte Cavalcante Barlavento por todos os pudins que alimentaram corpo e alma no decorrer desta jornada. A minha amada esposa, Deborah Bezerra Monteiro pela paciência e compreensão durante todo o curso. Aos professores de TADS pela paciência e por todo o conhecimento que me foi passado. Ao meu orientador Paulo Guedes por ter me aceitado como orientando e evitado que este trabalho se arrastasse para sempre. A todas as amizades feitas durante o curso, em especial a Yuri Rodrigues, Pedro Jatobá, Johnison Freitas, Edmilson Santana e Douglas Santana, que diversas vezes tiveram que segurar a "barra"e varar noites para concluir trabalhos. Ao pessoal do LIS-UFPE pela tolerância com os atrasos e ausências. A Maria Cecília Mota pelo computador utilizado nos testes.

RESUMO

A crescente complexidade dos sistemas de tempo real torna necessária a utilização de técnicas e ferramentas que possibilitem aos projetistas um maior controle das aplicações desenvolvidas, tornem o desenvolvimento estruturado, possibilitem a reutilização de código e proporcionem meios para a manutenção das aplicações. Os sistemas operacionais de tempo real existem para suprir estas necessidades, a maior parte desses sistemas são proprietários e possuem um custo de licenciamento alto. Devido à necessidade de desenvolver um sistema operacional de tempo real de baixo custo diversos projetistas criaram soluções que dessem ao Linux suporte para executar aplicações de tempo real. O *patch* Preempt_RT, suportado oficialmente pelos desenvolvedores do *kernel* Linux, e o RTAI, uma solução que utiliza uma arquitetura com dois *kernels* em paralelo, são soluções capazes de transformar o Linux em um sistema operacional de tempo real. Neste trabalho as duas soluções foram aplicadas a um sistema Linux que teve seu desempenho medido por meio de um conjunto de testes e os resultados avaliados, verificando a real capacidade do sistema em atender os requisitos de uma aplicação de tempo real.

Palavras-chave: Linux. Tempo Real. Sistemas Operacionais. Preempt_RT. RTAI.

ABSTRACT

The increasing complexity of real-time systems makes it necessary to use techniques and tools that allow designers greater control of the applications developed, make development structured, enable reuse of code and provide means for the maintenance of applications. Real-time operating systems exist to meet these needs, most of which are proprietary and have a high licensing cost. Due to the need to develop a low-cost real-time operating system, many designers have developed solutions that give Linux support for running real-time applications. The Preempt_RT patch, officially supported by Linux kernel developers, and RTAI, a solution that uses a parallel kernel architecture, are solutions that can turn Linux into a real-time operating system. In this work the two solutions were applied to a Linux system that had its performance measured through a set of tests and the results evaluated, verifying the real capacity of the system to meet the requirements of a real-time application.

Keywords: Linux. Real Time. Operating Systems. Preempt_RT. RTAI.

SUMÁRIO

1 INTRODUÇÃO

Sistemas de tempo real se tornaram elemento constante na vida das pessoas e estão presentes em locais que vão de aparelhos condicionadores de ar a grandes usinas geradores de energia.

Devido ao aumento da complexidade dos sistemas de tempo real, desenvolvedores de *software* procuram soluções que permitam a construção destes sistemas de forma rápida, estruturada e passível de manutenção a longo prazo, o que sempre foi uma grande dificuldade nos sistemas desenvolvidos em linguagem de montagem. Devido ao aumento constante da capacidade de processamento dos computadores, como parte da solução, sistemas operacionais de tempo real vem sendo cada vez mais utilizados no desenvolvimento de novos projetos, pois proporcionam uma grande variedade de funcionalidades previamente implementadas e facilidade no gerenciamento do *hardware*.

Grande parte dos sistemas operacionais de tempo real disponíveis são propriedade de empresas que os comercializam a um alto custo e, em alguns casos, esses sistemas não possuem suporte a diversos recursos avançados exigidos por algumas aplicações. Devido a esta situação, vários projetos foram desenvolvidos com a finalidade de transforma o Linux em um verdadeiro sistema operacional de tempo real. Dentre as soluções criadas podemos destacar o *patch* Preempt_RT e o *RealTime Application Interface* (RTAI).

A validação por meio da execução de *benchmarks,* da transformação do Linux em um sistema de tempo real utilizando Preempt_RT e RTAI, fornece uma base sólida de dados que permitem a um projetista de sistemas de tempo real comparar as soluções baseadas em Linux com outros sistemas e verificar se as restrições temporais de seus projetos podem ser atendidas.

Portanto este trabalho tem como objetivo avaliar a capacidade do *patch* Preempt_RT e RTAI de transformar o sistema operacional Linux num sistema operacional de tempo real capaz de tornar um computador de propósito geral do tipo *Personal Computer* (PC) com um único processador em um computador capaz atender aos requisitos de uma aplicação de tempo real rígida. Para esta avaliação foram escritas duas aplicações, baseadas nos testes realizados por (MOREIRA, 2007) e no *benchmark* Cyclictest (LINUX FOUNDATION, 2017). Neste processo foram realizadas as etapas: verificação da viabilidade de uso do *patch* Preempt_RT e do RTAI na máquina de testes e distribuição Linux escolhida, criação de *kernels* de tempo real utilizando as duas soluções estudadas, escrita dos testes e aplicação do testes, e avaliação dos resultados obtidos.

Durante a pesquisa da bibliografia foram identificados alguns trabalhos semelhantes e que proveram diversos recursos para a elaboração deste trabalho, dentre eles se destacam Moreira (2007), que forneceu o modelo de teste utilizado, assim como medições de performance do RTAI que foram utilizadas como valores de referência para verificação dos testes desenvolvidos e comparação com os valores obtidos.

Também podem ser destacados os trabalhos de Saoud (2011) e Hallberg (2017) que forneceram valores de referência e demonstram a real qualidade dos resultados oferecidos pelo *benchmark* Cyclictest na avaliação e comparação de sistemas Linux de tempo real.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Sistemas de tempo real

Quando falamos de Sistemas de Tempo Real (STR) logo se imagina algum sistema de controle industrial ou algum tipo de sistemas embarcado ultrarrápido, porém os STRs vão além destas aplicações, partindo, em seus primórdios computacionais, da decodificação de mensagens inimigas aos modernos sistemas computacionais utilizados por instituições financeiras que por questões regulamentares precisam garantir que os tempos de suas transações estejam dentro do limite estipulado sem, no entanto, serem longas ao ponto de provocarem perdas monetárias. Essa vasta gama de aplicações nos mostra o quão heterogêneos são estes sistemas e o quanto diversificada são suas implementações, variando de sistemas implementados utilizando alguma linguagem de montagem em microcontroladores de 8 bits a supercomputadores que executam complexos sistemas operacionais sobre os quais aplicações ainda mais complexas são também executas.

Um sistema pode ser dito de tempo real quando sua correção lógica está relacionada tanto a correção das saídas produzidas pelo sistema quanto pela sua pontualidade, ou seja, são sistemas que além de prezarem pela qualidade e correção dos seus algoritmos computacionais, também devem prezar pela pontualidade com que suas ações são tomadas, do contrário o sistema falhará (LAPLANTE, 2012).

Ao contrário do que diz o senso comum, um sistema de tempo real não tem como principais objetivos a diminuição dos tempos de resposta a estímulos ou simplesmente ser extremamente veloz, um sistema de tempo real tem como principal objetivo atender aos prazos estabelecidos no domínio do problema de forma determinística e por consequência previsível. Farines (2000) afirma que, um sistema de tempo real é previsível nos domínios lógico e temporal quando, independente das circunstâncias do *hardware*, carga ou falhas, pode-se saber com antecipação o seu comportamento antes da execução. A rigor, para que o comportamento do sistema seja estabelecido de forma determinística é necessário conhecer todas as variáveis que compõem o ambiente em que o sistema está inserido como: hipóteses de falha, carga computacional, arquitetura do *hardware*, sistema operacional, linguagem de programação, etc, e que em cada fase do seu desenvolvimento, metodologias e ferramentas sejam aplicadas na verificação do seu comportamento e previsibilidade.

2.1.1 Classificação

Existem diversas formas de se classificar os sistemas de tempo real, uma das formas mais comuns de fazê-lo, como descrito por Farines (2000) e Laplante (2012), é pela observação do rigor com que tratam seus requisitos temporais e no tipo de problemas que

falhas relacionadas a esses requisitos podem provocar. Sendo assim, os sistemas, podem ser classificados como sendo do tipo *soft* (brandos) ou *hard* (rígidos).

Um sistema de tempo real é brando quando o não cumprimento de suas restrições temporais provoca apenas alguma degradação no seu desempenho porém sem provocar falhas graves. Geralmente nesses sistemas os prejuízos provocados pela degradação do desempenho são compensados pelos benefícios de sua operação normal.

Um sistema de tempo real é rígido quando a falta no cumprimento de uma única restrição temporal pode levar a uma falha catastrófica. Nesse caso, uma falha no cumprimento de um prazo provoca prejuízos infinitamente maiores, como: uma catástrofe ambiental, a perda de vidas humanas ou grandes perdas materiais, que as benesses do sistema em operação normal.

Ainda é possível estabelecer uma terceira classificação entre os sistemas brandos e rígidos, o sistema de tempo real firme, onde se enquadram sistemas em que a perda de uma quantidade limitada de prazos não compromete o desempenho de forma crítica, porém a extrapolação do limite de prazos perdidos leva a uma falha catastrófica (LAPLANTE, 2012).

2.1.2 Tarefas de tempo real

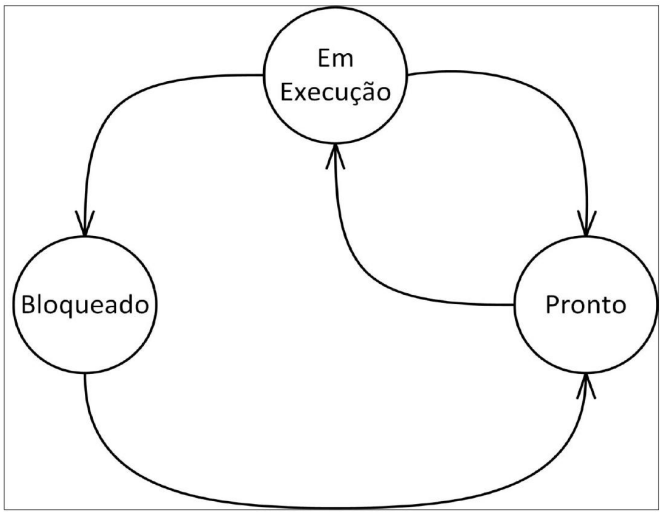
Como todo sistema computacional, STRs executam uma ou um conjunto de tarefas com o objetivo de produzir algum trabalho útil, ou podemos dizer simplesmente, que executam um programa sequencial, iniciando-se, normalmente, com a leitura de alguns dados de entrada e findando com a produção de algum resultado e alterando seu estado interno (KOPETZ, 2002).

Uma tarefa pode se classificada como de tempo real se obedece aos dois preceitos básicos definidos anteriormente para sistemas de tempo real, sua correção lógica está ligada a correção de suas saídas e a correção temporal. Tarefas de tempo real estão sujeitas ao cumprimento de prazos (*deadline*), e assim como os sistemas de tempo real, uma tarefa é dita crítica ou rígida, quando o não cumprimento de seus prazos pode provocar alguma falha catastrófica, e branda quando não lhes cumprir provocam, no máximo, uma diminuição do desempenho, sem grandes consequências.

As tarefas de tempo real, também são classificadas quanto a regularidade de sua ativação podendo ser periódicas, quando sua ativação ocorre em intervalos regulares predefinidos, ou como aperiódicas quando sua ativação ocorre de modo aleatório, normalmente em resposta a algum evento externo ou interno ao sistema. Quando executam múltiplas tarefas os sinais que determinam o início (ativação) e o término de uma tarefa são controlados pelo escalonador do sistema.

Quando um sistema executa múltiplas tarefas é necessária a implementação de um algoritmo de escalonamento. O escalonador é definido como o componente do sistema que é responsável pela implementação das políticas de acesso e gerenciamento de uso do processador pelas tarefas (FARINES, 2000). Quando o número de tarefas em execução simultânea é maior que o número de unidades de processamento disponíveis no sistema, além de um escalonador, o sistema deve prover algum mecanismo capaz de guardar o estado atual do processador associado as tarefas para que posteriormente esse estado possa ser carregado novamente e a execução retomada do ponto onde parou, este processo é chamado de chaveamento de contexto. Dentro deste cenário uma tarefa pode assumir os estados: Pronta, Em Execução e Bloqueada (TANENBAUM, 2009), estes estados se relacionam conforme a figura 1.

Figura 1: Estados de uma tarefa e seus relacionamentos



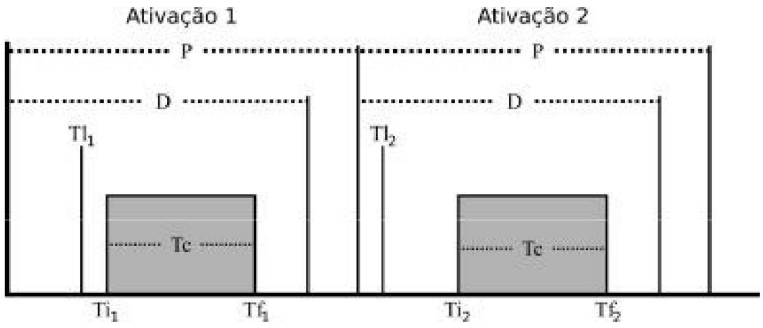
Fonte: adaptado de Tanenbaum (2009)

Na modelagem de tarefas de tempo real são utilizados alguns parâmetros que definem seu comportamento temporal: tempo de computação (Tc), tempo inicial (Ti), tempo final (Tf), tempo de liberação (Tl), *deadline* (D) e, caso a tarefa em questão seja periódica, o período (P).

O tempo de computação corresponde ao tempo total exigido para que a tarefa seja completada. Os tempos inicial e final correspondem aos instantes em que a tarefa inicia e finaliza, respectivamente, sua execução durante uma janela de ativação. O tempo de liberação é o momento em que uma tarefa entra no estado de "Pronto". O *deadline*, como já foi dito, é o prazo máximo que uma tarefa tem para concluir sua execução. E o período corresponde ao intervalo de tempo com que a tarefa repete sua execução. Outro parâmetro que deve ser considerado, principalmente quando múltiplas tarefas são executadas, e usado por diversos algoritmos de escalonamento, é a prioridade (PR), que representa a urgência relativa de

execução de uma tarefa em relação as outras. O conhecimento desses parâmetros é bastante importante para garantir a previsibilidade do sistema, além de possibilitarem a verificação da viabilidade, montar as tabelas e definir as políticas de escalonamento. Os parâmetros de tempo estão ilustrados na figura 2.

Figura 2: Características temporais de uma tarefa de tempo real



Fonte: figura extraída de Farines (2000)

2.2 Sistemas operacionais de tempo real

Com o aumento da complexidade das aplicações de tempo real (ATR) e consequentemente do *hardware* utilizado na execução das aplicações de tempo real, houve também um aumento na complexidade do trabalho realizado pelos desenvolvedores de aplicações. Para contornar parte desse problema a utilização de sistemas operacionais de tempo real (SOTR) tornou-se cada vez mais comum. Outro ponto importante, é o fato de que é bastante comum ATRs possuírem algumas funcionalidades que não possuem restrições temporais como interações com banco de dados, acesso à internet, interfaces gráficas, etc, e que sem o uso de um sistema operacional, implementar estes recursos é um processo bastante exaustivo.

Um sistema operacional (SO) é um dispositivo de *software* que tem como principal finalidade fornecer um ambiente em que certas funcionalidades do sistema possam ser gerenciadas de forma autônoma e oculta ao desenvolvedor, proporcionando uma camada de abstração com um conjunto próprio de instruções sobre a qual o desenvolvedor possa maximizar seu trabalho utilizando uma interface de programação mais amigável (TANENBAUM, 2009).

Estendendo essa definição aos STRs podemos dizer que um SOTR, além de possuir características comuns aos SOs, deve criar um ambiente de desenvolvimento previsível e determinístico qualquer que seja a carga imposta ao sistema. Um SOTR deve fornecer um ambiente no qual ATRs tenham seus requisitos temporais respeitados e executar de modo que suas ações possam ser previstas. Como geralmente são sistemas reativos, SOTRs devem

atender a requisitos relacionados a responsividade. Um SOTR deve garantir que respostas a estímulos, sejam internos ou externos, sempre serão dadas dentro de um intervalo de tempo que respeite os requisitos do sistema.

Assim como dito para os STRs, um SOTR não tem como principais objetivos a redução dos tempos de resposta a estímulos ou ter uma performance superior quando comparado a um Sistemas Operacionais de Propósito Geral (SOPG). SOTRs de qualidade podem ter desempenho global semelhante a SOPGs, porém o primeiro, normalmente, sacrificará a performance em detrimento da previsibilidade. SOPGs podem, em 99,9% dos casos, executar tarefas num tempo menor que SOTRs, todavia nos 0,1% dos casos restantes, o tempo de execução de uma tarefa será imprevisível, podendo ser até 1000 vezes mais longo que em um SOTR, isso seria mais que suficiente para reprovar o sistema em uma aplicação crítica. Embora a execução de tarefas em um SOTR possam ser executadas num tempo maior, são executadas com a garantia de estarem sempre dentro dos prazos estabelecidos.

Dentre as principais funções de um SOTR, está o fornecimento de mecanismos e ferramentas para que seja possível a execução de ATRs de modo previsível e satisfatório, atendendo aos requisitos impostos pelo domínio do problema. Os mecanismos oferecidos devem possibilitar ao desenvolvedor avaliar se o SOTR em questão é adequado ou não para a execução das aplicações pretendidas, isso inclui tornar acessível conhecer os valores de tempo máximo de execução de suas chamadas de sistema, os valores máximos das latências provocadas pelas operações internas do sistema e os valores de tempo relacionados as ATRs.

É importante observar que a qualidade dos valores apresentados está diretamente ligada a granularidade dos temporizadores disponibilizados pelo sistema, quanto maior for a resolução dos temporizadores melhor a qualidade das medições. Estes valores de tempo também podem variar de acordo com a arquitetura do processador em que o sistema é executado, com o código gerado pelo compilador utilizado para produzir o sistema, com os algoritmos utilizados nos processos internos do sistema e com a qualidade das suas respectivas implementações.

Alguns SOTRs, como o FREERTOS descrito em (BERRY, 2016), fornecem ao desenvolvedor apenas um conjunto mínimo de funcionalidades: suporte a multitarefa, escalonador com diversas políticas de escalonamento, *mutex*, semáforos e temporizadores. Estes pequenos SOTRs, também chamados de núcleos de tempo (NTR) real ou *real time kernel*, são pensados para serem pequenos, velozes e capazes de implementar políticas de tempo bastante rígidas. Suas características se aplicam muito bem a sistemas embarcados baseados em microcontroladores e que trabalham com restrições temporais bastante rígidas.

Os sistemas operacionais mais difundidos como: Linux, Windows e Mac OS X, oferecem outros recursos mais avançados: serviços de entrada e saída de dados, proteção de memória, sistemas de arquivos, políticas de segurança, interface com o usuário etc. Estes recursos tornam possível ao desenvolvedor construir e executar aplicações complexas que proporcionam um maior nível de segurança e interação com outros sistemas e usuários.

Originalmente estes sistemas não foram projetados para atender aos requisitos dos STRs, porém seus desenvolvedores vem implementando alternativas para este fim, como as diferentes versões do Windows Embedded e os *patchs* disponibilizados para Linux: Preempt_RT e RTAI. Vale lembrar que muitos dos sistemas em que se baseiam alguns SOTRs, como Linux, já são utilizados em várias aplicações de missão crítica, além de que sistemas de controle baseados em PC são uma realidade na indústria como complemento e até substitutos aos tradicionais sistemas de controle baseados em *relés*. Estes sistemas permitem uma maior flexibilidade na programação, expansão e configuração de *software* e *hardware*.

2.2.1 Latência

Segundo o dicionário Priberam (2017) latência é definida como: “tempo decorrido entre o estímulo e a resposta correspondente”.

No caso de sistemas computacionais e mais especificamente em SOs, um estímulo, pode ser externo, como o pressionar de um botão, ou interno, como uma *thread* que foi colocada no estado "pronto" e espera para ser executada.

Assim como todos os sistemas reais, SOTRs, estão sujeitos a latências que surgem como consequência do seu próprio funcionamento e do *hardware* sobre os quais executam. O conhecimento dos valores de latência e principalmente sua manutenção, mesmo em um sistema sobrecarregado, são primordiais na garantia da previsibilidade. O conhecimento dos valores de latência também são de vital importância na seleção de um SOTR que seja capaz de atender aos requisitos temporais de uma aplicação. São causas comuns de latência em SOs: latência de interrupção, latência de escalonamento, latência por inversão de prioridade, latência por inversão de interrupção (HART, 2007).

A latência de interrupção corresponde ao tempo decorrido entre a ocorrência de uma interrupção e o momento em que é atendida. Há também o tempo entre o atendimento da interrupção e a rotina que realmente processará o sinal recebido. O tempo decorrido entre o despertar de um processo de alta prioridade e a sua execução as vezes podem ser consideradas latência de interrupção, uma vez que o seu despertar muitas vezes ocorre devido a algum evento externo.

A latência provocada pelo escalonamento, é o tempo entre o instante em que uma tarefa de alta prioridade acorda (tempo de liberação) e o momento em que ela inicia a execução (tempo inicial).

As latências por inversão de prioridade e inversão de interrupção consistem no tempo que uma *thread* com prioridade alta espera para utilizar algum recurso em uso por uma thread de prioridade baixa, seja ela associada a uma outra tarefa ou, no caso da interrupção, a um manipulador de interrupções. Diferente da inversão de prioridade e inversão de interrupção não pode ser antecipada visto que a ocorrência da maior parte das interrupções não pode ser prevista.

2.2.2 Linux para tempo real

As modernas aplicações de tempo real estão muito mais conectadas e iterativas, isso exige a implementação de novas funcionalidades, como interfaces gráficas, comunicação com serviços web e banco de dados. Essas funcionalidades além de serem melhor desenvolvidas por meio da reutilização de código, também fazem necessário um melhor suporte dos sistemas operacionais sobre as quais executam. Alguns SOTR tradicionais e os núcleos de tempo real, embora tenham um ótimo desempenho no cumprimento de metas temporais, geralmente, oferecem pouco ou nenhum suporte aos recursos utilizados nas aplicações mais modernas.

Na busca por melhor suporte as aplicações, vários projetos tomaram a iniciativa de incorporar funcionalidades de tempo real aos SOPG. Devido ao seu código fonte aberto, sua comprovada robustez em aplicações críticas e sua portabilidade entre diferentes plataformas de *hardware*, o Linux, tornou-se um dos sistemas mais utilizados nas conversões para tempo real. Porém o Linux é concebido para uso em aplicações de propósito geral, executadas em computadores pessoais e servidores e por este motivo seu *kernel* é otimizado para obter um melhor desempenho global, alocando recursos de forma justa para todos os processos em execução.

Dentre os projetos mais ativos, no trabalho de transformar o Linux em um SOTR, podemos citar: RTAI, Xenomai e Preempt_RT. Cada um desses projetos implementa uma versão modificada do *kernel*, com arquitetura própria, vantagens e desvantagens. Neste trabalho as soluções estudadas e avaliadas são o RTAI e o *patch* Preempt_RT.

2.2.3 O *patch* PREEMPT_RT

O *patch* Preempt_RT é a solução de tempo real oficialmente suportada pelo *kernel* Linux. Este projeto é o mais bem sucedido no esforço para transformar o Linux em um SOTR sem o auxílio de um *microkernel*.

As modificações impostas pelo *patch* Preempt_RT ao *kernel* incluem, a transformação de alguns manipuladores de interrupção em *threads* de baixa prioridade (manipuladores de interrupção importantes para o funcionamento do sistema como um todo, como o temporizador, são mantidos com prioridade máxima), substituição das *spin_locks* por *mutexes* para permitir que as regiões críticas do *kernel* passem a ser preemptíveis e herança de prioridade. Algumas alterações não são mais necessárias pois foram incorporadas ao ramo principal do *kernel*, como temporizadores de alta resolução e *mutexes* no espaço de usuário.

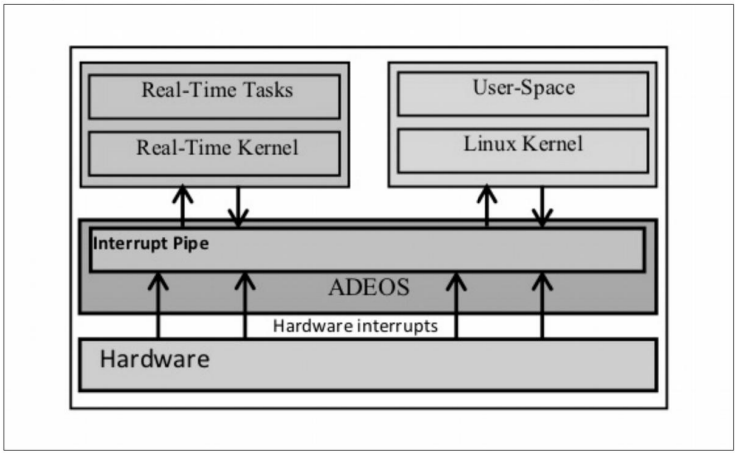
As funcionalidades do sistema permanecem idênticas para funções de gerenciamento, comunicação e sincronia entre *threads*, porém são adicionados três novos algoritmos de escalonamento: *First In First Out* (FIFO) orientado a prioridades (em que 99 é a prioridade máxima), *Round Robin* (RR) e *Earliest Deadline First* (EDF).

2.2.4 O RTAI

O RTAI, acrônimo de *Real-time Application Interface*, surgiu como uma variação do antigo RTLinux desenvolvida pelo *Dipartimento di Ingeneria Aerospaziale* da Universidade Politecnico di Milano.

Sua abordagem para a transformação do Linux em um SOTR utiliza um *microkernel* em paralelo ao *kernel* do Linux que é responsável pela execução das tarefas de tempo real, enquanto o *kernel* fica responsável pela execução das tarefas de baixa prioridade, esta solução é comumente chamada de *Dual Kernel*. Neste sistema o gerenciamento do *hardware* fica a cargo de uma camada de abstração criada abaixo dos *kernels*, chamada ADEOS (*Adaptive Domain Environment for Operating Systems*). Esta camada é quem permite aos dois núcleos compartilharem o mesmo *hardware*. A arquitetura implementada pelo sistema é ilustrada na figura 3.

Figura 3: Arquitetura *Dual Kernel* adotada pelo RTAI.



Fonte: figura extraída de Saoud (2011)

O *microkernel* implementado pelo RTAI, a semelhança dos NTR, suporta todo o conjunto básico de funcionalidades para a construção de ATR, como rotinas para a criação, destruição, suspensão, sincronia e comunicação entre tarefas, temporizadores com alta resolução, políticas de escalonamento, capacidade de execução de tarefas de tempo real no espaço do usuário e mecanismos para controle de recursos compartilhados. Seu escalonador suporta políticas *Rate Monotonic* (RM), EDF, FIFO orientado a prioridades (em que 0 é a maior prioridade) e RR.

3 METODOLOGIA

A avaliação de um SOTR é guiada principalmente pela verificação da capacidade de suas características atenderem aos requisitos de um determinado projeto, o que pode envolver diversas variáveis que influenciam o desempenho do sistema em várias circunstâncias diferentes. Diversas características relacionadas a requisitos não funcionais de uma aplicação também podem ter peso maior ou menor na avaliação de um SOTR, tais como: suporte e reputação dos desenvolvedores, documentação, custo, integração com sistemas legados, suporte a *hardware* específicos etc, corroboram com o número de fatores que tornam a comparação entre SOTRs um processo complexo.

As avaliações de desempenho de SOTR mais completas, normalmente são baseadas na observação do sistema como aplicação destinada a fins específicos. Estas avaliações são difíceis de generalizar e portar para outras soluções ou que, simplesmente, possuam arquitetura de destino diferente da proposta nos testes originais. Pesquisas vem sendo desenvolvidas na tentativa de criar métodos genéricos de avaliação com a escolha de parâmetros quantitativos e qualitativos que sejam comuns à maior parte dos sistemas de tempo real, e que estejam diretamente relacionados aos principais casos em que se aplicam. Estes métodos de avaliação são classificados quanto ao grau de detalhamento com que observam os sistemas testados, segundo Schwan (1994), em: *Fine-grained benchmarks*, *Application-oriented benchmarks* e *Simulation-based evaluations*.

Os *Fine-grained benchmarks* analisam a execução de STR observando suas características de baixo nível, analisando o desempenho do conjunto *hardware* e *software.* Este tipo de análise confere excelente precisão aos resultados obtidos porém requer um alto grau de conhecimento sobre o funcionamento do *hardware* assim como o analisar por meio de instrumentos de medição específicos.

Benchmarks Simulation-based evaluations, utilizam a execução de modelos, com um grau de detalhes considerado suficiente para a aplicação estudada, do sistema avaliado. Este método possui a vantagem de poder executar testes em sistemas, tanto de *hardware* como *software*, que ainda não estejam completamente implementados. No entanto os resultados obtidos possuem um grau de precisão proporcional a qualidade do modelo avaliado e a quantidade de variáveis, que alteram o funcionamento do sistema.

Os *Application-oriented benchmarks*, observam a execução de STR por meio de parâmetros de alto nível como: cumprimento de *deadlines*, tempo de execução das tarefas e latências, relacionados a uma aplicação sintética com requisitos relacionados a execução semelhantes ao de aplicações reais. Este método de análise tem como principal vantagem

observar o comportamento do conjunto *hardware* e *software*, porém sem que seja possível atribuir com precisão as causas dos valores medidos.

3.1 Modelagem dos testes

Este trabalho utiliza uma abordagem de testes do tipo *Application-oriented benchmarks*, concebidos para verificar a capacidade do *patch* Preempt_RT e do RTAI em transformarem um sistema Linux de propósito geral num SOTR a partir da medida dos valores de latência, do tempo de computação das tarefas executadas e da verificação do cumprimento dos *deadlines* estabelecidos.

O modelo de *benchmark* implementado foi baseado na solução proposta por Moreira (2007) junto com o algoritmo de medição de latências do programa Cyclictest (LINUX FOUNDATION, 2017). Em seu trabalho, Moreira (2007), propõe duas aplicações de teste que une as definições e conceitos operacionais dos *benchmarks* MiBench (BROWN *et al.*, 2001) e Hartstone (KAMENOFF; WEIDERMAN, 1992). Em ambas as aplicações é proposto que cada uma das tarefas que as compõe execute uma função dentre as sugeridas por *benchmarks* MiBench e Moreira (2007) para simulação de aplicações automotivas e de controle industrial, embora vários produtos de consumo e de uso hospitalar também implementem funções semelhantes. As funções implementadas neste trabalho foram:

- Multiplicação de duas matrizes 5 x 5
- Ordenação de 20 inteiros usando *quicksort*
- Transformação de graus para radianos
- Cálculo da raiz quadrada de inteiros utilizando séries de Taylor
- Cálculo polinomial cúbico

Dentre as duas aplicações teste desenvolvidas, a primeira utiliza a definição do *benchmark* Hartstone para a Série-PH, tarefas periódicas e harmônicas, e simula o comportamento do que poderia ser um programa responsável pelo monitoramento de um conjunto de sensores com taxas de amostragem diferentes e sem a intervenção de interrupções ou do usuário. Na aplicação são implementadas cinco *threads* periódicas com frequência igual a um múltiplo de todas as outras frequências maiores, no caso dos testes implementados as frequências foram as mesmas sugeridas nas definições do Hartstone: 1Hz, 2Hz, 4Hz, 8Hz e 16Hz. Cada *thread* executa uma das funções descritas anteriormente, o *deadline* para a execução das tarefas foi definido como sendo igual aos seus respectivos períodos e prioridade de execução máxima para todas as tarefas.

A segunda aplicação executa, além das cinco *threads* da primeira, mais duas *threads* aperiódicas, simulando aplicações que precisam responder a eventos provocados por interrupções, conforme a definição do *benchmark* Hartstone para Série-AH. O intervalo de ativação das *threads* aperiódicas foi gerado de forma aleatória dentro de um intervalo de 20ms a 40ms, seus *deadlines* foram definidos em 20ms, e a função executada por elas foi a conversão de graus para radianos.

Além das funções listadas acima, as tarefas, tanto periódicas quanto aperiódicas, foram responsáveis pela execução das medições das suas próprias latências, tempo de computação e verificação do cumprimento de *deadline*.

3.2 Implementação dos testes

Os testes foram implementados em linguagem C, linguagem para a qual são fornecidas as bibliotecas tanto do RTAI quanto as chamadas de sistema e bibliotecas fornecidas pelo Linux, utilizadas com o Preempt_RT.

A construção dos programas foi realizada seguindo algumas premissas sobre os sistemas que executariam as aplicações de teste. Foi levado em conta o suporte a temporizadores de alta resolução, funções cujo tempo de execução são previsíveis, isolamento entre as tarefas, execução das aplicações em modo usuário, suporte a mecanismos que evitem paginação, cache e memória em disco. O atendimento de todas estas premissas foi feito por ambas as soluções, seja pela utilização de funções do próprio sistema Linux, seja pelo provimento de uma biblioteca específica.

Por necessidade das aplicações desenvolvidas para RTAI, que precisam utilizar funções de tempo real fornecidas por bibliotecas próprias e não podem executar chamadas de sistema dentro do código de tempo real, foram desenvolvidos quatro programas de teste: Série-PH e AH para Preempt_RT e, Série-PH e AH para RTAI.

Todos os programas seguiram a mesma sequência básica de funcionamento descrita abaixo, o fluxograma contendo o fluxo básico de funcionamento do programa pode ser visualizado no apêndice B:

1. Alocação de memória
2. Travamento das posições de memória atuais e futuras, para que permaneçam na memória RAM
3. Configuração das *threads* como tarefas de tempo real, com a definição da prioridade e do algoritmo de escalonamento (FIFO).
4. Execução das *threads* e suas respectivas computações

Também foram implementados nos programas de teste, um mecanismo que permite a todas as *threads* de tempo real iniciarem a execução o mais próximas possível e um mecanismo para a impressão de histogramas contendo a distribuição dos resultados de medição das latências.

3.3 O ambiente de testes

Na comparação entre diferentes SO, e mais especificamente de SOTR, é importante que a configuração do *hardware* utilizado nos testes propostos seja igual ou no mínimo equivalente, isso garante que os resultados obtidos sejam consistentes e que não tenham sido influenciados por funcionalidades específicas de uma determinada configuração de *hardware*.

O *hardware* utilizado para testar as duas soluções de tempo real escolhidas foi um *netbook* Acer, modelo Aspire One D250-1023, processador com arquitetura x86, Intel Aton N270, *clock* de 1,60GHz, memória *cache* L2 de 512KB, 1GB de memória DDR2-533, disco rígido de 320GB SATA.

Antes da escolha de um *hardware* para a execução de qualquer uma das soluções estudadas é importante observar algumas particularidades relacionadas a possíveis conflitos entre as configurações do *kernel* e o *hardware* que foram verificadas neste trabalho e explicadas mais adiante.

Ambas as soluções de tempo real testadas usam como base o sistema operacional Linux e a distribuição escolhida foi Debian 8.8 (Jessie) para processadores de 32 bits. A distribuição Debian foi escolhida, dada a facilidade de se produzir um sistema com funcionalidades reduzidas, sua ampla documentação, sua grande coleção de pacotes contendo programas e bibliotecas pré compilados e por ser a base de inúmeras outras distribuições que se aplicam de servidores a sistemas embarcados.

Foi considerado de grande importância produzir *kernels* com configurações idênticas, com óbvia exceção às opções específicas exigidas por cada uma das soluções, para que o máximo de recursos não alterassem o desempenho dos sistemas de forma a favorecer alguma das soluções testadas. As configurações utilizadas tiveram como ponto de partida a configuração *vanilla* de cada *kernel*. A versão utilizada do *patch* PREEMPT_RT foi a 4.4.17-rt25 publicada em 25 de agosto de 2016, aplicado sobre um *kernel, vanilla*, versão 4.4.17. A versão testada do RTAI foi a 5.0.1 publicada em 15 de maio de 2017, o *patch* HAL foi aplicado em um *kernel*, *vanilla*, versão 4.4.43. Vale mencionar que não existem versões do *kernel* que sejam suportadas por ambas as soluções simultaneamente.

3.4 Produzindo um *kernel* de tempo real

Para produzir um sistema Linux de tempo real utilizando uma das soluções estudadas neste trabalho não é necessário nenhum truque especial. Seguem-se todos os passos de construção de um *kernel* de propósito geral: configuração, compilação, instalação dos módulos e instalação do *kernel*, porém é necessária uma atenção especial durante o processo de configuração.

A construção de um SOTR baseado em Linux utilizando as duas soluções estudadas se inicia com a aplicação dos respectivos *patchs* ao *kernel*. Estes *patchs* alteram algumas características e adicionam novas funcionalidades ao *kernel*.

As principais mudanças promovidas pelo *patch* Preempt_RT é tornar o *kernel* completamente preemptível, inclusive em sua região crítica, ao substituir as *spin_locks* por *mutexes*, transformar os tratadores de interrupção, com exceção de interrupções vitais para o sistema como a dos temporizadores, em *threads* com prioridade menor que as tarefas de tempo real, evitando assim o problema da latência provocada pela inversão de interrupção, e a implementação de uma política de herança de prioridade, minimizando o problema da inversão de prioridade.

Já o *patch* HAL, fornecido junto com o RTAI, é menos invasivo e apenas adiciona ao *kernel* suporte a camada de abstração de *hardware* ADEOS. Esta camada de abstração permite que dois *kernels* funcionem simultaneamente compartilhando o mesmo *hardware*.

3.4.1 Configuração do *kernel*

Como dito anteriormente, este processo requer uma atenção especial, pois é nesta etapa que identificaremos as funcionalidades do *kernel* que provocam alterações nos valores de latência, por vezes os deixando imprevisíveis. Para auxiliar neste processo foi utilizado o programa Cyclictest.

O Cyclictest é um programa fornecido junto a suíte de teste *rt-tests* que por sua vez é fornecida e oficialmente mantida pelo grupo de desenvolvimento do *kernel*. Sua principal função é a medição do conjunto de latências provocadas pelo sistema ou por outros processos a que estão submetidas um conjunto de tarefas. Cyclictest proporciona uma poderosa funcionalidade que detecta processos executados pelo sistema que provoquem distúrbios nos valores de latência.

As várias execuções do programa Cyclictest mostraram que os recursos do *kernel* voltados ao gerenciamento de energia foram os que mais interferiram nos valores de latência. Recomenda-se que serviços como: *Advanced Conguration and Power Interface* (ACPI), *CPU*

Frequency Scaling e *CPU Idle*, sejam desabilitados. É importante notar que isso é uma grande restrição ao uso das soluções estudadas em sistemas alimentados por bateria.

Algumas outras funcionalidades foram desabilitadas e habilitadas tendo como princípio exigências e características das soluções de tempo real abordadas como: temporizadores com alta resolução, carregamento de módulos etc. Outras foram alteradas a fim de produzir um *kernel* sem otimizações que privilegiassem algum tipo de arquitetura de *hardware* específica, numa tentativa de tornar o sistema o mais genérico possível.

A habilitação e remoção de recursos do *kernel*, podem produzir efeitos indesejados e conflitos com algumas configurações de *hardware*. Alguns *notebooks* não suportam a inicialização de um *kernel* sem suporte a ACPI ou a incompatibilidade do RTAI com o suporte a recursos usados por alguns processadores AMD inviabilizam o uso do sistema nesta plataforma. Infelizmente nenhuma das duas soluções possui alguma documentação sobre o *hardware* suportado, reforçando a importância da execução de testes.

3.4.2 RTAI: instalação, compilação e execução de programas

Após a configuração, compilação e instalação do *kernel* de tempo real, diferente do Preempt_RT, o RTAI precisa ser configurado e instalado. O processo é semelhante a instalação de programas em Linux a partir do código fonte.

Outra exigência do RTAI é que para a compilação e execução de programas é necessário definir as variáveis CFLAGS e LDFLAGS, estas variáveis são utilizadas no processo de compilação e definem os locais onde se encontram as bibliotecas utilizadas na construção dos programas. Para que os programas possam ser executados os módulos do RTAI precisam ser carregados. Como os módulos possuem uma hierarquia de dependência é preciso ativá-los em uma ordem específica. Este processo foi automatizado por meio de um *script* para *shell* (*bash*) reproduzido na figura 9 do apêndice C.

3.5 Execução dos testes

Os testes foram executados sem maiores problemas e os sistemas responderam de forma adequada, sem quebras ou travamentos.

Para que os resultados pudessem servir como valores de referência e para que as reais capacidades dos sistemas testados pudessem ser avaliadas, foram executados dois procedimentos para sobrecarregar os recursos da máquina de testes conforme o proposto por Lim (2017). Um dos procedimentos, que consiste na execução infinita do comando *ping*, produz uma utilização de 100% do processador, o outro processo foi uma compilação do *kernel* Linux, que além de provocar uma sobrecarga no processamento, também provoca um

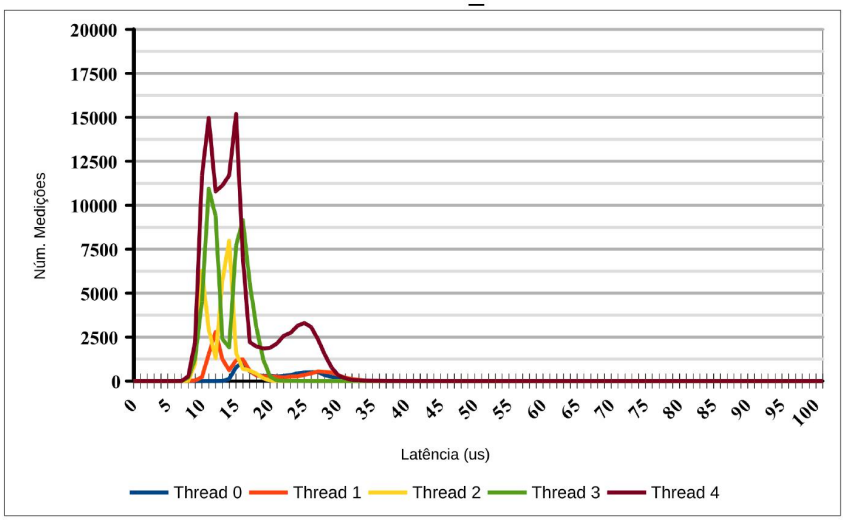
intenso uso de memória RAM e do disco rígido. Para que os valores medidos fossem consistentes os testes foram executados por um período de duas horas.

Após a execução, foram plotados gráficos dos histogramas gerados, contendo os valores de latência medidos para cada uma das *threads* executadas, e os valores máximos de computação de cada tarefa executada foram compilados em uma tabela. Os dados de saída dos testes são analisados no capítulo 4.

4 ANÁLISE DOS RESULTADOS

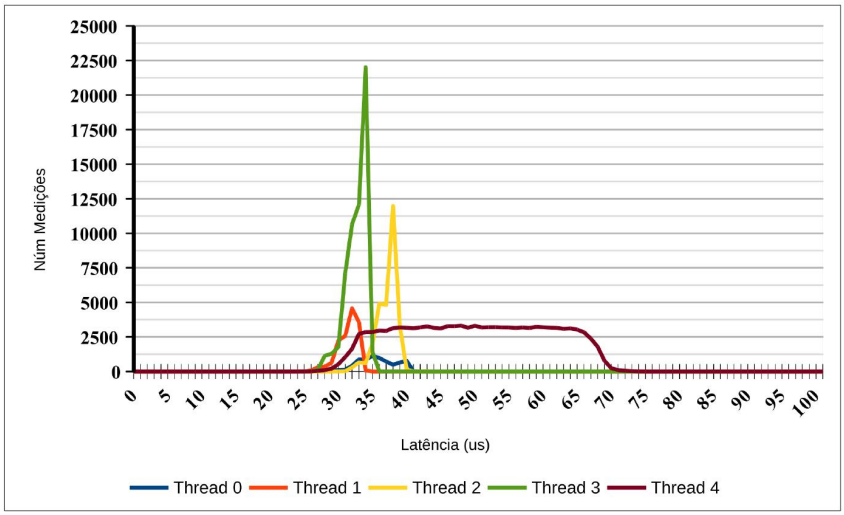
Os testes para Série-PH nos mostram intervalos de latência bastante consistentes com diferença entre os extremos suficientemente restrita, como nos mostra a tabela 1. Podemos dizer que quanto mais restrito o intervalo de valores mais previsível o sistema, como pode ser visto nas tarefas executadas tanto com o Preempt_RT (figura 4), que registrou latências menores, quanto com RTAI, intervalos mais estreitos (figura 5), com exceção da *thread* 4 do teste executado no RTAI na qual fica evidente a existência de uma anomalia, e que não teve sua causa investigada neste trabalho..

Figura 4: Medidas de latência aferidas durante a execução do teste Série-PH com o PREEMPT_RT



Fonte: o autor

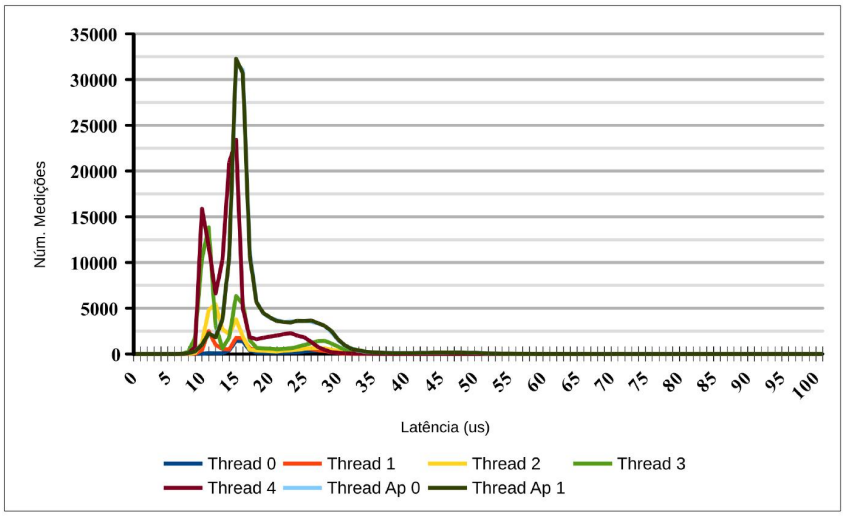
Figura 5: Medidas de latência aferidas durante a execução do teste Série-PH com o RTAI



Fonte: o autor

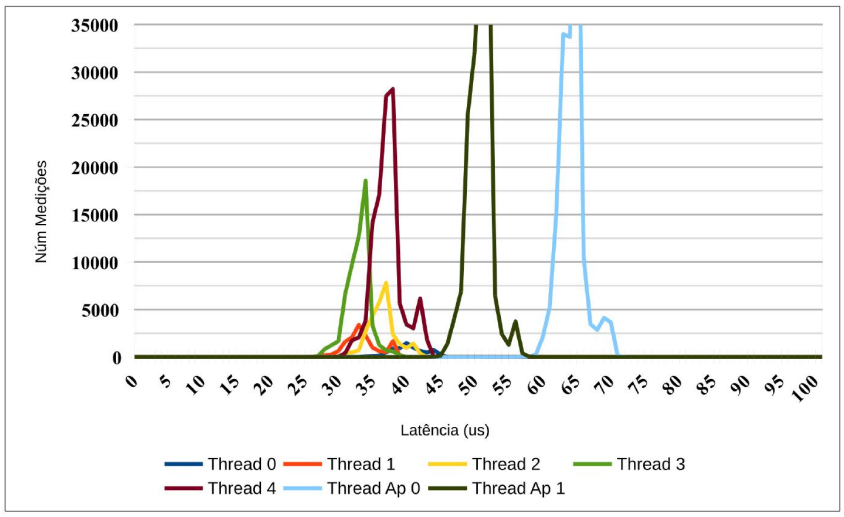
Embora estejam distribuídos de forma adequada, os valores máximos de latência, em alguns casos, superam 100% do tempo de computação máximo das tarefas (tabela 3) o que pode ser um grande problema para tarefas com deadlines na casa dos microssegundos, porém para as tarefas executadas, a soma dos valores de Latência e Tempo de Computação foram bem inferior aos deadlines definidos.

Figura 6: Medidas de latência aferidas durante a execução do teste Série-AH com o PREEMPT_RT



Fonte: o autor

Figura 7: Medidas de latência aferidas durante a execução do teste Série-AH com o RTAI



Fonte: o autor

Quando adicionadas duas tarefas aperiódicas aos testes (Série-AH) e observados os histogramas das figuras 6 e 7 assim como a tabela 2, podem ser visualizados alguns comportamentos interessantes. A execução das tarefas pelo *patch* Preempt_RT a primeira vista

se mostraram inalteradas, mas uma análise detalhada dos valores de latência mostram alguns pontos fora da curva e registros de latência máxima bem superiores a maioria das medições feitas nos testes da Série-PH, embora os valores não tenham comprometido a execução da aplicação, a soma dos valores de latência e tempo de computação ainda foram bem inferiores ao *deadline*, esse tipo de comportamento imprevisível reforça a necessidade de testes de medição de latência com a aplicação pretendida.

A coluna com os valores medidos no teste com a Série-AH da tabela 3 nos mostra que os tempos de computação para o Preempt_RT foram praticamente o dobro dos valores medidos para os testes com a Série-PH, no caso da *thread* 1 o valor foi quase 4 vezes maior, embora mais uma vez os deadlines foram respeitados. Já o RTAI, não demonstrou alterações significativas nos valores de latência e nos valores do tempo de computação para os testes com a as Série-AH e Série-PH (tabela 3), porém na figura 7 podemos ver um comportamento que tende a adiar a execução das tarefas aperiódicas ao longo do tempo, embora os valores tenham estado dentro de um intervalo bem definido e as curvas serem muito parecidas, podemos nos questionar se a adição de novas tarefas aperiódicas provocaria o aumento das latências destas tarefas.

A análise dos valores medidos para latências a que as tarefas de tempo real estão sujeitas e dos seus respectivos tempos de computação nos mostram que, tanto o Preempt_RT quanto o RTAI, podem executar com segurança, tarefas de tempo real com restrições temporais na casa dos milissegundos. Para sistemas que possuam tarefas com restrições temporais menores que 1ms é recomendada, além da execução de testes de validação dentro da própria aplicação, o projeto cuidadoso do escalonamento das tarefas, com especial atenção a atribuição das prioridades, caso seja utilizado um algoritmo de escalonamento FIFO ou RM.

Ao compararmos os valores obtidos com os resultados apresentados na tabela 4, fornecidos por Moreira (2007), que aplicou testes semelhantes ao RTAI, e Saoud (2011), que utilizou o *benchmark* Cyclictest para medir a latência no Preempt_RT, podemos dizer que, embora tenham sido um pouco mais altos, mostram que o comportamento dos sistemas ante a heterogeneidade dos métodos de análise, testes aplicados e *hardwares* utilizados, esteve dentro do esperado e com uma variação de valores relativamente pequena, o que seria esperado já que as principais funcionalidades de um SOTR é abstrair o comportamento do *hardware* e apresentar ao desenvolvedor uma camada de abstração consistente, independente da situação em que se encontra o sistema, sobre a qual possam desenvolver suas aplicações de forma previsível.

Tabela 1: Valores (em µs) máximos e mínimos de latência obtidos nos testes Série-PH - Preempt_RT x RTAI

	Preempt_RT		RTAI	
Thread	Latência Máx.	Latência Mín.	Latência Máx.	Latência Mín.
0	39	13	41	28
1	43	8	38	22
2	30	8	44	27
3	25	7	40	24
4	44	7	74	20

Fonte: o autor

Tabela 2: Valores (em µs) máximos e mínimos de latência obtidos nos testes Série-AH - Preempt_RT x RTAI

	Preempt_RT		RTAI	
Thread	Latência Máx.	Latência Mín.	Latência Máx.	Latência Mín.
0	70	9	46	32
1	72	8	39	25
2	71	7	43	28
3	74	7	39	21
4	67	7	45	23
Ap. 0	80	6	72	52
Ap. 1	71	7	59	37

Fonte: o autor

Tabela 3: Valores (em µs) do tempo de computação máximo obtidos nos testes Serie-PH e Serie-AH - Preempt_RT x RTAI

	Preempt_RT		RTAI	
Thread	Tc (Série-PH)	Tc (Série-AH)	Tc (Série-PH)	Tc (Série-AH)
0	18	70	19	21
1	38	81	45	45
2	39	83	37	36
3	37	70	28	20
4	39	79	43	40
Ap. 0	-	62	-	42
Ap. 1	-	56	-	44

Fonte: o autor

Tabela 4: Valores (em µs) de latência e tempo de computação obtidos por Moreira (2007) e Saoud (2011)

RTAI (MOREIRA, 2007)		Preempt_RT (SAOUD, 2011)
Latência	Tc	Latência
3	34 a 47	7 a 62

Fontes: o autor, valores obtidos de (MOREIRA, 2007; SAOUD, 2011)

5 CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma análise quantitativa do desempenho de duas soluções, *patch* Preempt_RT e RTAI, capazes de transformar um sistema Linux de propósito geral em um SOTR. Os resultados desta análise, os programas de teste desenvolvidos e a documentação gerada contribuem com informações valiosas para projetistas de STR e estudantes no momento de comparar outros SOTR com os sistemas estudados assim como base para a criação de ATR utilizando Linux.

Além dos resultados obtidos com os testes, em uma avaliação qualitativa do *patch* Preempt_RT, podemos destacar sua grande facilidade de utilização dada a sua boa documentação, mantida atualizada e organizada. As aplicações de tempo real são fáceis de construir pois utilizam as mesmas bibliotecas já suportadas e documentadas pelo Linux. A sua principal deficiência é o aparecimento de alguns valores espúrios de latência que podem comprometer o desempenho de algumas aplicações.

Quanto ao RTAI, a maior consistência nos valores de latência e tempo de execução permitem produzir sistemas mais previsíveis, porém vários problemas de suporte e compatibilidade, como documentação escassa (algumas funcionalidades sequer estão documentadas), dispersa e desatualizada, arquitetura eficiente mas que lhe confere pouca integração com o Linux e a impossibilidade de executar chamadas de sistemas em tarefas de tempo real sob pena de tornar a execução do sistema imprevisível, podem tornar seu uso inviável para projetos que exijam um bom suporte ou uma aplicação que precise de uma maior integração com o Linux.

Como sugestão para investigações futuras seria bastante desejável comprovar a eficiência do Preempt_RT e RTAI por meio de um prova de conceito em uma aplicação prática como em um sistema de controle. Como um dos algoritmos de escalonamento para tarefas de tempo real, o EDF é suportado tanto pelo Preempt_RT quanto pelo RTAI, testar a eficiência dos sistemas utilizando este algoritmo seria de grande importância. Com a popularização de processadores com múltiplos núcleos se torna inevitável o estudo do comportamento de SOTR nessas plataformas. Como também se tornaram popular, seria de grande interesse estudar o comportamento de um sistema Linux de tempo real em plataformas utilizadas em dispositivos embarcados baseadas em processadores ARM. Avaliar a necessidade e a possibilidade de executar o Linux com a aplicação do *patch* Preempt_RT e do RTAI simultaneamente com o objetivo de tentar sanar deficiências de ambas as soluções.

REFERENCIAS

BERRY, Richard. **Mastering the FreeRTOS real time kernel**. [*s.l.*]: Real Time Engineers, 2016.

BROWN, Richard B *et al*. MiBench: A free, commercially representative embedded benchmark suite. *In:* Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. **Anais**[…] Austin, 2001, v. 1, n. 1, p. 3-14.

FARINES, J., FRAGA, J.S. e OLIVEIRA, R.D. **Sistemas de tempo real**, Florianópolis: Departamento de Automação e Sistemas da Universidade Federal de Santa Catariana, 2000.

LINUX FOUNDATION. **Cyclictest**. Disponível em: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest. Acesso em: 3 ago. 2017.

HALLBERG, Andréas. **Time critical messaging using a real-time operating system**. 2017. Dissertação (Mestrado em Ciência da Computação) - Chalmers University of Technology, University of Gotherburg, Gotemburgo, 2017.

HART, Darren V. Internals of the RT Patch. *In:* Proceedings of the Linux Symposium. **Anais**[…] Ottawa, 2007**,** v. 1, p. 161-171.

KAMENOFF, Nick I., WEIDERMAN, Nelson H. Hartstone uniprocessor benchmark: Denitions and experiments for real-time systems. **Real-Time Systems**, [*s.l.*] , v. 4, n. 4, 1992, p. 353-382.

KOPETZ, Hermann. **Real-time systems - design principles for distributed embedded applications**. [*s.l.*]: Kluwer Academic, 2002.

LAPLANTE, Phillip A. **Real-time systems design and analysis**. [*s.l.*]: Wiley, 2012.

LIM, Geunsik. **Worst case latency test scenarios**. 2017. Disponível em: https://wiki.linuxfoundation.org/realtime/documentation/ howto/tools/worstcaselatency. Acesso em: 3 ago. 2017.

MANTEGAZZA, Paolo. **RTAI 3.4 user manual rev. 0.3.** [*s.l.*]: [*s.n.*], 2006.

MOREIRA, Andeson Luiz Souza. **Análise de sistemas operacionais de tempo real**. 2007. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal De Pernambuco, Recife, 2007.

PRIBERAM. Latência. **Priberam da Língua Portuguesa**. Disponível em: https://www.priberam.pt/dlpo/. Acesso em: 17 out. 2017.

SAOUD, Slim Ben. Impact of the linux real-time enhancements on the system performances for multi-core intel architectures. **International Journal of Computer Applications**, v. 17, n. 3, 2011.

SCHWAN, Karsten. **A survey of real-Time operating systems**. [*s.l.*]:[*s.n.*], 1994.

TANENBAUM, Andrew S. **Sistemas operacionais modernos**. 3 ed. São Paulo: Pearson Education do Brasil, 2009.

APÊNDICE A – PRINCIPAIS OPÇÕES ALTERADAS NA CONFIGURAÇÃO DO *KERNEL*

As configurações requeridas pelo Preempt_RT e pelo RTAI estão marcadas com seus respectivos nomes.

A opção *Fully Preemptible kernel* (RT) só está disponível após a aplicação do *patch* Preempt_RT.

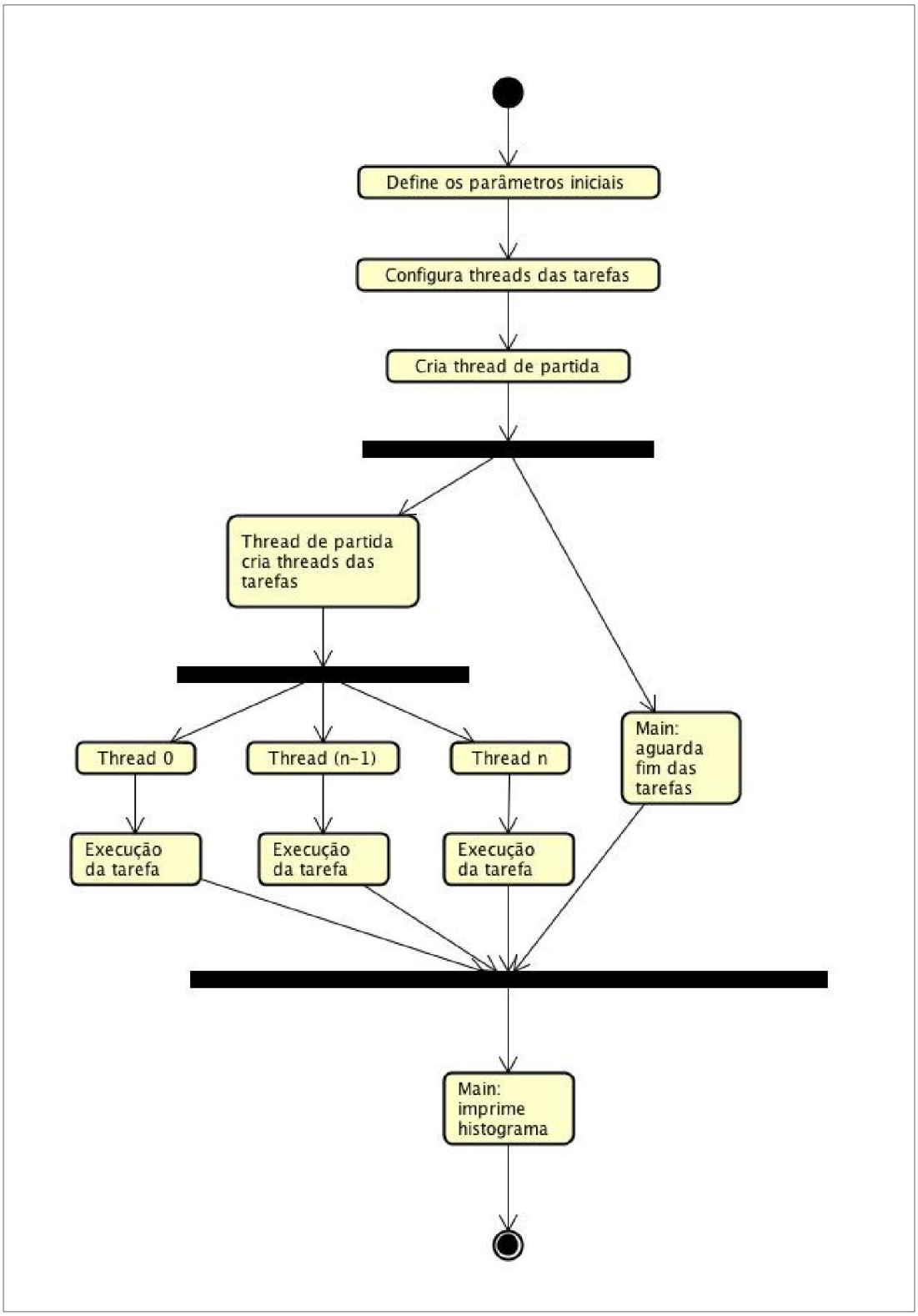
A opção *Interrupt pipeline* só está disponível após a aplicação do *patch* HAL do RTAI.

O arquivo de configuração utilizado tem como base a versão *vanilla* dos *kernels* utilizados.

- *Kernel* 32 bits
  - [ ] 64-bit *kernel*
- *General setup* > *Timers subsystem*
  - [*] *High Resolution Timer Support*
  - [*] *Enable Loadable module support* (RTAI)
  - [ ] *Module versioning support* (RTAI)
- *Processor type and features*
  - [ ] *Symmetric multi-processing support* (SMP)
  - *Processor family* > (X) *Pentium-Classic*
  - *Preemption Model* > (X) *Fully Preemptible kernel* (RT) (Preempt_RT)
  - [*] *Interrupt pipeline* (RTAI)
  - *Time frequency* > (X) 1000HZ
  - [ ] AMD *MCE features* (RTAI)
- *Power Management and ACPI options*
  - [ ] ACPI (*Advanced Conguration and Power Interface*) *Support*
  - CPU *Frequency scaling* > [ ] CPU *Frequency scaling*
  - CPU *Idle* > [ ] CPU *Idle PM support*
- *File systems* > *Pseudo filesystems*
  - [*] */proc file system support* (RTAI)
- *Kernel hacking*
  - [ ] *Debug preemptible kerne*l
  - [ ] *Debug the x86 FPU code*

APÊNDICE B – FLUXOGRAMA DOS PROGRAMAS DE TESTES

Figura 8: Visão geral do fluxo de execução dos programas de teste



Fonte: o autor

APENDICE C – *SCRIPT* PARA INICIALIZAÇÃO DO RTAI

Figura 9: *Script* para inicialização do RTAI (rtai-init.bash)

```
#!/bin/bash

#rtai-init.bash
#Carrega os módulos do RTAI
sudo insmod /usr/realtime/modules/rtai_hal.ko
sudo insmod /usr/realtime/modules/rtai_sched.ko
sudo insmod /usr/realtime/modules/rtai_fifos.ko
sudo insmod /usr/realtime/modules/rtai_sem.ko
sudo insmod /usr/realtime/modules/rtai_mbx.ko
sudo insmod /usr/realtime/modules/rtai_msg.ko
sudo insmod /usr/realtime/modules/rtai_shm.ko
sudo insmod /usr/realtime/modules/rtai_smi.ko
sudo insmod /usr/realtime/modules/latency_rt.ko

#Criação das variáveis para compilação
export CFLAGS=$ (/usr/realtime/bin/./rtai-config
--lxrt-cflags)
export LDFLAGS=$ (/usr/realtime/bin/./rtai-config
--lxrt-ldflags)
```

Fonte: o autor

Para que as variáveis criadas pelo *script* possam ser utilizadas por todos os usuários é preciso utilizar a notação ponto seguido por espaço:

Figura 10: Comando para execução do *script:*

```
$ . rtai-init
```

Fonte: o autor

APÊNDICE D – ESTRUTURA BÁSICA DE UM PROGRAMA UTILIZANDO PREEMPT_RT

Figura 11: Exemplo de código utilizando PREEMPT_RT

```
/*
* Exemplo de program de tempo real usando PREEMPT_RT
* Executa uma computação simples em tempo real
*/

#include <stdlib.h>
#include <time.h>
#include <sched.h>
#include <sys/mman.h>
#include <string.h>

/* define a prioridade do processo (maior: 99, menor: 0) */
#define PRIORIDADE (99)

#define MAX_PROG_PILHA (8*1024)

/*
*     Configura a prioridade de um processo e define a política de
escalonamento
* Retorna 1 em caso de falha e 0 para sucesso
* Para processos de tempo real "pri" deve ter um valor entre 0 e 99
* O parâmetro "pol" deve ter um entre os cinco valores de finidos para as
* políticas de escalonamento definidas em sched.h
*/
int setPrioridade( int pri, int pol );

/*
*     Inicializa a pilha do programa com zeros
* O parâmetro tp define o tamanho da pilha
*/
void stackPrefalt( int tp );

/*
*     Garante que a porção de memoria reservada para o processo não será
alterada nem paginada
*     Inicializa as posições de memória com 0
* Retorna 1 em caso de falha e 0 para sucesso
* O parâmetro tPilha deve ter o tamanho da pilha do processo que se
deseja
* bloquear. No linux a pilha de um processo normalmente tem 8MB de
tamnho.
*/
int travaMemoria( int tPilha );
```

Figura 11: continuação

```
int main( int argc, char** argv ) {

      /* Inicializa processo como de tempo real */

      if( setPrioridade( PRIORIDADE, SCHED_FIFO ) == 1  ) {
      perror( "Falha na configurcao do escalonador" );
      exit( -1 );
      }

      if( travaMemoria( MAX_PROG_PILHA ) == 1 ) {
      perror( "Falha na alocacao de memoria" );
      exit( -2 );
      }

      /* código da computação aqui! */

      exit(0);
}

int setPrioridade( int pri, int pol ) {
      struct sched_param param;
      param.sched_priority = pri;

      if( sched_setscheduler( 0, pol, &param ) == -1 ) {
      return 1;
      } else {
      return 0;
      }
}

void stackPrefault( int tp ) {
      unsigned char pilha[ tp ];
      memset( pilha, 0, tp );
}

int travaMemoria( int tPilha ) {
      if( mlockall( MCL_CURRENT | MCL_FUTURE ) == -1 ) {
      return 1;
      } else {
      stackPrefault( tPilha );
      return 0;
      }
}
```

Fonte: o autor

APÊNDICE E – ESTRUTURA BÁSICA DE UM PROGRAMA UTILIZANDO RTAI

Figura 12: Exemplo de código utilizando RTAI

```
/*
*     Exemplo de program de tempo real usando PREEMPT_RT
* Executa uma computação simples em tempo real de forma periódica
*/

#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <signal.h>
#include <sched.h>
#include <rtai_lxrt.h>

/* Tempo que a tarefa demora para iniciar a execução */
#define DELAY (0)

#define INTERVALO (1000000) // Intervalo de 1 milissegundo

int main( int argc, char *argv[] ) {
      unsigned long long int numMed = 0;
      int i = 0;
      RTIME t1, t2, exectime[3];
      RT_TASK *task;

      /*
      * Inicia o timer com o parâmetro 0, isso deve ser feito pois
estamos usando
      * o modo oneshot e o valor do período não é relevante.
      */
      start_rt_timer( 0 );

      /* Trava as posições de memória */
      mlockall( MCL_CURRENT | MCL_FUTURE );

      /* Cria uma nova thread de t1 real */
      if( !( task = rt_task_init_schmod( nam2num( "MAINB" ), 0, 0, 0,
SCHED_FIFO, 0xF ) ) ) {
      printf( "Tarefa nao pode ser criada.\n" );
      exit( 1 );
      }

      /*
      *     Torna a tarefa atual uma tarefa de tempo real e
      * diz que sua execução  periÃ³dica
      */
      rt_make_hard_real_time();
      rt_task_make_periodic_relative_ns( task, DELAY, INTERVALO );

      /* Inicializa o escalonador do RTAI */
      rt_set_oneshot_mode();

      /* Código de computação aqui! */

      stop_rt_timer();
      rt_make_soft_real_time();

      if( task ) {
      rt_task_delete( task );
      }

      return 0;
}
```

Fonte: o autor