CAPÍTULO 1

PLATAFORMA OPERACIONAL

1.1 INTRODUÇÃO

Nos capítulos que precedem, o protocolo *DoRiS* foi descrito e os seus objetivos foram apresentados. Depois desta fase de definição e validação do protocolo *DoRiS*, é chegado o momento de apresentar um outro desafio deste projeto de desenvolvimento de um protocolo de comunicação de tempo real baseado em Ethernet. Isto é, a escolha uma plataforma operacional de tempo real híbrida, de código livre, que possa ser utilizada tanto em computadores de uso geral quanto em dispositivos dedicados e que atenda os requisitos temporais necessários para a implementação do protocolo *DoRiS*.

É importante notar aqui que o desafio principal concentra-se em torno do uso dos computadores de uso geral. De fato, os dispositivos dedicados utilizam sistemas embarcados ou micro-controladores para oferecer um conjunto de funcionalidades definidas. Integrar tais dispositivos num sistema distribuído requer levar em conta as suas restrições específicas como, por exemplo, capacidade de processamento, taxa de transferência, consumo de energia, etc. No entanto, a qualidade dos serviços oferecidos por tais dispositivos é uma conseqüência do seu projeto. Conseqüentemente, uma vez especificados os requisitos do sistema, o dispositivo pode ser escolhido adequadamente de tal forma a satisfazer estes requisitos.

No caso dos computadores de uso geral, a situação é bastante diferente, pois o primeiro objetivo a ser alcançado é o desempenho geral do sistema. No entanto, as tecnologias modernas utilizadas nas arquiteturas de computadores atuais para aumentar o desempenho dos dispositivos de hardware são muitas vezes fontes de imprevisibilidade temporal. Isto é, por exemplo, o caso da memória *cache*, do acesso direto a memória (DMA), do co-processamento, da predição de instruções, das unidades *multicore* ou dos *pipelines*. Devem também ser mencionadas as funções de gerenciamento da energia, que, por afetar a freqüência de execução do processador, dificultam a estimativa dos piores casos dos tempos de execução dos programas. Tal comple-

xidade do hardware, embora torne os sistemas computacionais mais velozes, aumentam o grau de imprevisibilidade do sistema e dificultam a estimativa dos atributos necessários à teoria do escalonamento de Sistemas de Tempo Real (LIU; LAYLAND, 1973; AUDSLEY et al., 1993; TINDELL; BURNS; WELLINGS, 1994). No entanto, reduzir o conjunto de serviço oferecidos por um Sistema Operacional de Propósito Geral (SOPG) no patamar dos SOs que podem garantir previsibilidade equivale a negar as evoluções do hardware das últimas décadas. Para resolver este desafio, várias abordagens foram propostas ao longo do tempo para desenvolver plataformas operacionais determinísticas baseadas em SOPG.

Este capítulo apresenta algumas destas abordagens com o objetivo de escolher a plataforma operacional que oferecerá suporte à implementação de *DoRiS*. Como esta deverá ser de software livre, usar a família de sistemas Linux parece ser uma tendência natural. Inicialmente, uma explicação geral sobre as principais características de Linux é apresentada na seção 4.2. Como será visto, Linux oferece vários aspectos quem impedem seu uso direto na implementação de *DoRiS*. Tais aspectos serão detalhados na seção 4.3. Em seguida, algumas das tendências que têm incorporado características de sistemas de tempo real em Linux serão descritas na seção 4.4. Por fim, resultados comparativos de algumas plataformas serão apresentados na seção **??** e, baseados na necessidade de *DoRiS*, a escolha de uma destas será justificada.

1.2 LINUX: UM SISTEMA OPERACIONAL DE PROPÓSITO GERAL

Depois de uma breve justificativa da escolha do Linux para este trabalho, esta seção apresenta os principais elementos do sistema Linux que causam ou que são relacionados à existência de imprevisibilidade temporal na execução das aplicações.

1.2.1 Motivação para o uso de Linux

A escolha da plataforma operacional no contexto deste trabalho se deu pelas características do protocolo *DoRiS* e pelo contexto universitário do seu desenvolvimento. Resumidamente, os seguintes critérios foram adotados para determinar uma plataforma adequada para desenvolvimento de *DoRiS*:

i) Ser de código livre e aberto (licença GPL).

ii) Garantir desvios máximos da ordem da dezena de micro-segundos para as aplicações de controle via rede, conforme o padrão de qualidade de serviço oferecida pelos Sistemas Operacionais de Tempo Real (SOTR).

iii) Permitir o uso de aplicações multimídia, banco de dados e outros componentes de software distribuídos tipicamente usadas em ambientes multi-usuários de Sistemas Operacionais de Propósito Geral (SOPG).

iv) Prover suporte às aplicações embarcadas usando componentes de prateleira.

A primeira destas condições decorre do modelo de pesquisa e desenvolvimento defendido pelo autor destas linhas. Apesar da ecologia política ser um assunto essencial aos olhos deste mesmo, foge do escopo deste trabalho apresentar os elementos referindo a este tópico. A leitura de André Gorz (GORZ, 1977) ou Milton Santos (SANTOS, 2000) deverá saciar a curiosidade dos mais interessados.

A segunda condição decorre dos requisitos temporais das aplicações de controle e corresponde também às margens de desvios necessárias para a implementação do protocolo *DoRiS* conforme visto no capítulo . Finalmente, os terceiro e quarto ítens são diretamente relacionados com as metas do protocolo *DoRiS*, que já foram amplamente discutidas no capítulo .

Dentre as soluções de SOPG, o sistema Linux (L. Torvalds et al., 2008; BOVET, 2005) é um software livre e de código aberto que se distingui pela originalidade da sua proposta de desenvolvimento sob licença GNU/GPL (GNU General Public License). Um conseqüência direta deste modelo de desenvolvimento cooperativo é que o *kernel* Linux conta com uma das maiores comunidades de desenvolvedores espalhada pelo mundo inteiro. Outra vantagem do *kernel* Linux é ser baseado no sistema UNIX e oferecer uma interface de utilização conforme o padrão POSIX *Portable Operating System Interface* (IEEE, 2004). Por fim, deve ser observado que Linux é bastante difundido nos ambientes de pesquisa acadêmica.

Por todas estas razões, a plataforma Linux apareceu como uma candidata pertinente para este projeto de pesquisa. No entanto, como será visto mais adiante, o *kernel* Linux por si só não oferece as garantias temporais definidas no ítem (iv) acima. Portanto, revelou-se necessário pesquisar soluções existentes para tornar o SOPG Linux determinista. Algumas destas soluções serão apresentadas ao longo deste capítulo, assim como a extensão Linux de tempo real adotada no contexto deste trabalho.

Antes de abordar a caracterização das propriedades temporais do Linux e de algumas das suas extensões de tempo real, ir-se-á apresentar brevemente alguns conceitos básicos de sistemas operacionais que serão utilizados intensivamente ao longo deste capítulo.

1.2.2 Interrupções

Na classe do sistema do tipo UNIX (BACH, 1986), na qual se encontra o sistema Linux, o SO, ou simplesmente, *kernel*, executa num modo diferente dos demais processos: o modo "protegido". Diz-se que os processos, associados às aplicações, executam em modo "usuário". Estes dois modos são definidos no nível do hardware do processador e são utilizados para restringir o acesso aos dispositivos de hardware da máquina. Por este meio, garante-se que os únicos processos que podem obter acesso aos dispositivos de hardware são aqueles executando em modo protegido. Responsável pelo gerenciamento dos recursos, o *kernel* provê uma camada de abstração dos dispositivos de hardware aos processos usuários. Na prática, a interface do Linux é oferecida através de trechos de código denominadas "chamadas de sistema". Cada uma destas chamadas oferece uma API (Interface de Programação da Aplicação) padronizada pela norma POSIX (IEEE, 2004), para facilitar tanto o trabalho dos programadores, quanto a portabilidade dos códigos de software.

A organização da comunicação entre os dispositivos, o *kernel* e as aplicações é baseada no conceito de interrupção do processador. Tais interrupções são gerenciadas por um dispositivo de hardware específico, o PIC ou o APIC (*Advanced Programmable Interrupt Controller*) que é diretamente conectado ao processador. Na ocorrência de um evento de comunicação, ou seja, quando um dispositivo requer a atenção do processador, ele informa o APIC via uma linha de interrupção (*IRQ lines*). Por sua vez, o APIC informa o processador que uma interrupção precisa ser tratada.

Distingui-se as interrupções síncronas e assíncronas. As primeiras são geradas pelo próprio processo em execução no final do ciclo de execução de uma instrução. Elas correspondem ao uso de instruções específicas pelo programa, tais como as chamadas de sistema da interface do *kernel*, ou podem ser causadas pela execução de uma instrução indevida. As interrupções assíncronas são geradas pelos dispositivos de hardware para informar ao processador a ocorrência de um evento externo e podem ser geradas a qualquer instante do ciclo do processador.

Quando o processador percebe uma interrupção, quer seja síncrona ou assíncrona, ele desvia sua execução e passa a executar o tratador de interrupção associado à linha de interrupção que solicitou o APIC inicialmente. Esta execução não é associada a processo algum, mas sim à ocorrência de um evento e, portanto, não tem contexto de execução próprio. O tratador simplesmente executa no contexto do último processo que estava executando no processador.

Para minimizar o impacto das interrupções sobre a execução dos processos regulares, um tratador de interrupção é geralmente subdividido em três partes. A primeira parte executa operações críticas que não podem ser atrasadas e que modificam estruturas de dados compartilhadas pelo dispositivo e o *kernel*. Tais operações são executadas imediatamente e com as interrupções desabilitadas. Também executado imediatamente pelo tratador, mas com as interrupções habilitadas, são as operações rápidas que modificam apenas as estruturas de dados do *kernel*, pois estas são protegidas por mecanismos de *locks*, assim como será vista na seção 4.2.4.1. Estes dois conjuntos de operações constituem a parte crítica do tratador, durante a qual a execução não pode ser suspensa. Finalmente, as operações não-críticas e não-urgentes são possivelmente adiadas e executadas com as interrupções habilitadas. Estas execuções são chamadas de `softirqs` ou `tasklets`.

Do tratamento eficiente das interrupções depende a capacidade do sistema para reagir a eventos externos. O *kernel* padrão garante esta eficiência, proibindo que a execução da parte crítica do tratador de interrupção seja suspensa, mas permitindo que a parte não-crítica, e geralmente mais demorada, seja suspensa para a execução da parte crítica de uma outra interrupção.

1.2.3 Tempo compartilhado

O principal objetivo de um Sistemas Operacional de Propósito Geral (SOPG), tal como Linux, é oferecer o melhor serviço possível para o uso compartilhado por vários usuários de recursos limitados, tal como processadores, memória, disco, placas de rede e outros dispositivos de *hardware* (BACH, 1986; TANENBAUM, 2001; OLIVEIRA; CARISSIMI; TOSCANI, 2001). Um usuário do sistema, compartilhando um conjunto de recursos com outros, deverá ter a ilusão que ele está sozinho atuando naquele sistema. O SOPG deve, portanto, garantir o acesso dos usuários a todos os recursos que ele gerencia com latências imperceptíveis para um ser humano. A implementação de tal serviço, cuja qualidade depende altamente da subjetividade de cada

usuário e das aplicações que ele precisa executar, utiliza o mecanismo de compartilhamento temporal dos recursos. Para dar a impressão de exclusividade e continuidade dos serviços aos usuários, os recursos são alocadas sucessivamente às aplicações por fatias de tempos curtas, da ordem de alguns milisegundos. A alternância rápida destas alocações garante que cada aplicação ganhe o acesso ao recurso com uma freqüência suficiente para que não haja tempo ocioso perceptível do ponto de visto do usuário. O modelo de tempo compartilhado caracteriza assim os sistemas multi-programáveis e multi-usuários.

No Linux e em SOPG similares, o entrelaçamento das execuções dos processos ao longo do tempo é realizado da seguinte maneira. O tempo é dividido em intervalos de tamanho iguais. Para este efeito, o SO utiliza o PIT (*Programable Interrupt Timer*), baseado, nas arquiteturas PCs i386, num de hardware dedicado - o chip 8254 ou seu equivalente. O PIT é programado para gerar uma interrupção periódica, o *tick*, cujo o período, chamado de $jiffy$, é configurável, variando entre 1 e 10 μs em função das arquiteturas.

A cada *tick* uma interrupção ocorre. Esta provoca a atualização e execução eventual dos temporizadores do sistema assim como a chamada do escalonador quando isto se faz necessário. Conseqüentemente, quanto menor o $jiffy$, mais freqüentes serão as ativações de cada processo, melhorando assim a capacidade de reação do sistema. Por outro lado, com um $jiffy$ pequeno, a alternância entre os processos são mais freqüentes, aumentando o tempo gasto em modo *kernel*, no qual o processador só executa tarefas de gerenciamento. Portanto, escolher a freqüência dos *ticks* constitui um compromisso entre o desempenho do sistema e a resolução desejada para escalonar os processos. Observa-se, em particular, que a implementação do tempo compartilhado por *ticks* de duração constante faz com que um processo não possa "dormir" por um tempo menor do que um *jiffy*.

Além disso, o algoritmo de gerenciamento dos temporizadores, chamado de "roda dos temporizadores", pode constituir uma outra fonte de sobrecarga nos sistemas que utilizam muitos temporizadores. Este algoritmo utiliza uma estrutura de armazenamento baseada em 5 faixas de $jiffies$ correspondendo a intervalos de tempo que crescem exponencialmente (BOVET, 2005; MOLNAR, 2005). Cada faixa armazena os temporizadores de acordo com os seus valores de instantes de expiração. A primeira faixa corresponde aos temporizadores com tempos de expiração contidos no intervalo indo de 1 a 256 $jiffies$. A segunda corresponde aos temporizadores expirando entre 257 $jiffies$ e 16384 $jiffies$ e assim por diante até a quinta faixa que corresponde

a todos os temporizadores expirando depois de $67108865\,jiffies$. Quando um temporizador é criado, ele é armazenado na faixa que contém o $jiffy$ no qual ele deve expirar. A cada 256 $jiffies$, depois que todos os temporizadores da primeira faixa sejam eventualmente disparados, o sistema executa a rotina chamada "cachoeira" que atualiza as diferentes faixas, cascateando os temporizadores de faixa em faixa. Por exemplo, esta rotina determina quais são os temporizadores da segunda faixa que vão expirar nos próximos 256 $jiffies$ e os transferem para a primeira faixa. Similarmente, a rotina transfere os devidos temporizadores da terceira para a segunda faixa, da quarta para terceira e da quinta para a quarta.

Além de utilizar uma estrutura de dados de tamanho limitado, este algoritmo se torna muito eficiente quando os temporizadores são apagados antes de serem disparados. Isto ocorre, por exemplo, no caso de uma estação servidor Internet na qual a grande maioria dos temporizadores são cancelados rapidamente. Neste caso, a sobrecarga causada pela execução da rotina da cachoeira passa a ser insignificante, pois os temporizadores são cancelados antes de serem cascateados. Por outro lado, percebe-se que a diminuição da duração do $jiffy$ aumenta a sobrecarga gerada por este algoritmo, pois a rotina da "cachoeira" acaba sendo executada mais freqüentemente. Além disso, sendo o tempo entre cada execução menor, o número de temporizadores ainda não cancelados é maior. Conseqüentemente, o número de transferências de faixa a ser realizado para cada temporizador, antes do seu cancelamento eventual, aumenta.

Na sua publicação inicial, o *kernel* 2.6 passou a usar um $jiffy$ de $1ms$. Percebeu-se então que este valor gerou uma sobrecarga significativa no sistema, notadamente devido aos temporizadores utilizados pelas conexões TCP. Este fato explica em parte porque as versões do *kernel* posteriores a 2.6.13 vêm com o valor padrão do $jiffy$ de $4ms$, e não mais de $1ms$.

1.2.4 Preempção

No contexto dos escalonadores baseados em prioridade, um processo de baixa prioridade pode ser suspenso no decorrer da sua execução para ceder o processador a uma processo de prioridade mais alta. Quando tal evento ocorre, diz-se que houve preempção do processo em execução pelo processo de mais alta prioridade. De forma geral, diz-se que houve preempção de um processo A por um processo B, quando o processo A deve interromper sua execução para ceder o processador ao processo B. No caso dos processos executando em modo usuário,

a preempção corresponde a alternância de processos que o *kernel* executa para a implementação do mecanismo de tempo compartilhado. Este procedimento se torna mais complexo quando se trata da preempção de processos executando em modo *kernel*.

Diz-se, de maneira simplificada, que o *kernel* é preemptivo se uma alternância de processos pode acontecer quando o processador está em modo protegido. Considere, por exemplo, um processo A executando um tratador de exceção associado a uma chamada de sistema. Enquanto A está executando, o tratador de uma interrupção de hardware acorda um processo B mais prioritário que A. Num *kernel* não preemptivo, o *kernel* completa a execução da chamada de sistema de A antes de entregar o processador para o processo B. No caso de um *kernel* preemptivo, o *kernel* suspende a execução da chamada de sistema para começar a executar B imediatamente. Eventualmente, depois da execução de B, o processo A será escalonado novamente e a chamada de sistema poderá ser finalizada.

O principal objetivo de tornar o SO preemptivo (BOVET, 2005) é diminuir o tempo de latência que o processo de mais alta prioridade pode sofrer antes de ganhar o processador. Ver-se-á que este objetivo é de suma importância para que um SOPG tal como Linux possa oferecer as garantias temporais encontradas em STR.

1.2.4.1 Concorrência e sincronização Um dos desafios em tornar o *kernel* preemptivo é garantir a integridade dos dados, mesmo que vários caminhos de controle do *kernel* possam ter acesso aos mesmos dados de forma concorrente. Este é um problema fundamental e é comumente conhecido como problema da exclusão mútua (DIJKSTRA, 1965; LAMPORT, 1974; RAYNAL, 1986; MELLOR-CRUMMEY; SCOTT, 1991; LAMPORT; MELLIAR-SMITH, 2005). No entanto, já que as soluções adotadas pelo *kernel* fazem parte das principais causas de imprevisibilidade temporal do Linux padrão, vale a pena apresentar brevemente estas soluções.

Chama-se região crítica, qualquer recurso ou estrutura de dados que só pode ser utilizado por um processo (ou um conjunto de processos) de forma atômica. Isto é, se um processo P entra numa região crítica, nenhum outro processo pode entrar nesta mesma região crítica enquanto P não a liberou. Uma maneira simples de garantir a exclusão mútua num sistema monoprocessado é desabilitar a possibilidade de preempção durante a execução de uma região crítica. Esta solução, bastante utilizada nas versões do *kernel* não superiores à versão 2.4, tem dois inconvenientes relevantes. Primeiro, ela só funciona em sistemas monoprocessados

e, segundo, ela não impede o acesso da região crítica por tratadores de interrupção. Neste segundo caso, para garantir a exclusão mútua, um processo entrando numa região crítica que é compartilha com tratadores de interrupções deve também desabilitar aquelas interrupções.

Nas suas versões mais recentes, a partir da versão 2.6, o *kernel* tenta reduzir ao máximo o uso de tais soluções, que comprometem o desempenho do sistema em termos de capacidade reativa. No entanto, existem situações nas quais o uso destes mecanismos é necessário. Para este efeito, a implementação da exclusão mútua em contextos multiprocessados e/ou em tratadores de interrupções utiliza duas primitivas básicas de sincronização: os semáforos e os *spin-locks*.

1.2.4.2 Semáforos Um semáforo (TANENBAUM, 2001; BOVET, 2005) é constituído de um contador, de duas funções atômicas `up` e `down` e de uma fila. Quando um processo P quer entrar numa região crítica, ou de forma equivalente, quando P quer adquirir um recurso compartilhado que só pode ser adquirido simultaneamente por um número limitado de processos, ele chama a função `down` que decrementa o contador do semáforo. Se o valor resultante é positivo ou nulo, o processo adquiriu o semáforo. Senão, ele é suspenso depois de ter sido colocado na fila de espera. Após um processo terminar de usar o recurso, ele executa a função `up` que incrementa o valor do contador e acorda o primeiro processo da fila. O valor inicial n do contador define o número máximo de processos que podem adquirir este semáforo simultaneamente. No caso $n = 1$, o semáforo é simplesmente chamado de *mutex*.

Observa-se que o tempo que um processo fica suspenso, esperando por um semáforo, é imprevisível. Ele depende de quantos processos já estão esperando por aquele semáforo e do tempo que cada um deles permanecerá na região crítica. Além disso, um processo é autorizado a dormir enquanto ele está em posse de um semáforo. Estes fatos fazem com que os tratadores de interrupção que não são autorizados a dormir não possam usar semáforos.

Um outro ponto importante a ser observado é a ausência de "dono" do semáforo na implementação pelo *kernel* padrão. Isto é, quando um processo P adquire um semáforo, ele se torna "dono" deste. Mas esta informação não é disponível para os demais processos que possam tentar adquirir o semáforo enquanto P o detém. As conseqüências deste aspecto de implementação serão discutidas na seção 4.3.4

1.2.4.3 Spin-lock Nos ambientes multiprocessados, o uso de semáforo nem sempre é eficiente. Imagine o seguinte cenário: um processo P_1 executando num processador Π_1 tenta adquirir um mutex S, que já foi adquirido pelo processo P_2 que executa num outro processador Π_2. Conseqüentemente, P_1 é suspenso e o *kernel* entrega no processador Π_1 para um outro processo P_3. Mas, se a estrutura de dados protegida por S for pequena, P_2 pode executar a função up antes mesmo que P_3 comece a executar. Se P_1 é mais prioritário que P_3, uma nova alternância será realizada pelo *kernel* para permitir que P_1 volte a executar no processador Π_1. Percebe-se então que quando o tempo de execução de uma troca de contexto é maior do que o tempo de execução na região crítica, o uso de semáforos deve ser evitado. Nestes casos, a solução é utilizar os mecanismos de trancas e de espera ocupada fornecidos pelos *spin-locks*.

No *kernel* padrão, um *spin-lock* é uma variável booleana que é utilizada de forma atômica. Só um processo pode adquirir um *spin-lock* num dado instante. Quando um processo tenta adquirir um *spin-lock* que já está em posse de um outro processo, ele não é suspenso mas sim executa uma espera ocupada (*spin*), tentando periodicamente adquirir o *lock*. Isto evita as trocas de contextos do cenário acima. Como uma região crítica protegida por um *spin-lock* há de ser curta, um processo que adquire um *spin-lock* não pode ser suspenso. Portanto, a preempção é desabilitada enquanto um processo está em posse do *lock*. Além disso, quando um caminho de controle do *kernel* C utiliza um *spin-lock* que pode ser adquirido por um tratador de interrupção, ele precisa desabilitar as interrupções. Caso contrário, se uma interrupção ocorre enquanto C está em posse do *lock*, o tratador causa a preempção do processo C e fica em espera ocupada, tentando adquirir o *lock* que C detém. Mas, como o processador está ocupado pelo tratador, C não pode mais executar, e portanto, não pode devolver o *lock*, resultando no *deadlock* do sistema. Para impedir que tal cenário aconteça, o *kernel* adota a solução de desabilitar as interrupções durante um *spin-lock* que pode ser adquirido por um tratador de interrupções. Apesar de resolver o problema, esta solução pode aumentar significativamente o tempo de resposta do sistema na ocorrência de uma interrupção.

Observa-se que a implementação de *spin-locks* no *kernel* padrão não utiliza fila e que, assim como os semáforos, os *spin-locks* não tem “donos”.

1.3 CARACTERIZAÇÃO DO COMPORTAMENTO TEMPORAL DO LINUX

Esta seção apresentada as diferentes métricas adotadas para caracterizar o comportamento temporal SOPG Linux.

1.3.1 Precisão temporal de escalonamento

De acordo com (PICCIONI; TATIBANA; OLIVEIRA, 2001), o mecanismo de compartilhamento do tempo (discutido na seção 4.2.3) limita a resolução temporal disponível para escalonar os processos, utilizando o escalonador do *kernel* , a até duas vezes o valor do $jiffy$. Para observar este fato, definiu-se aqui o seguinte cenário ilustrado na figura 4.1. Seja ε e δ dois números arbitrários positivos e menores que $jiffy$. No instante T_1, um *tick* ocorre. Logo em seguida, no instante $t_1 = T_1 + \delta$, o processo corrente P executa a chamada de sistema `sleep` pedindo para dormir durante um intervalo de tempo $jiffy + \varepsilon$. O escalonador do *kernel*, que não trabalha com frações de $jiffy$, arredonda este valor para $2jiffy$, o múltiplo logo superior. Além disso, este tempo só começa a ser descontado a partir do instante do próximo *tick* T_2. Portanto, o temporizador associado ao `sleep` acorda P no instante $T_2 + 2jiffy$. Ao final, P dormiu $3jiffy - \delta$ ao invés de dormir $jiffy + \varepsilon$.

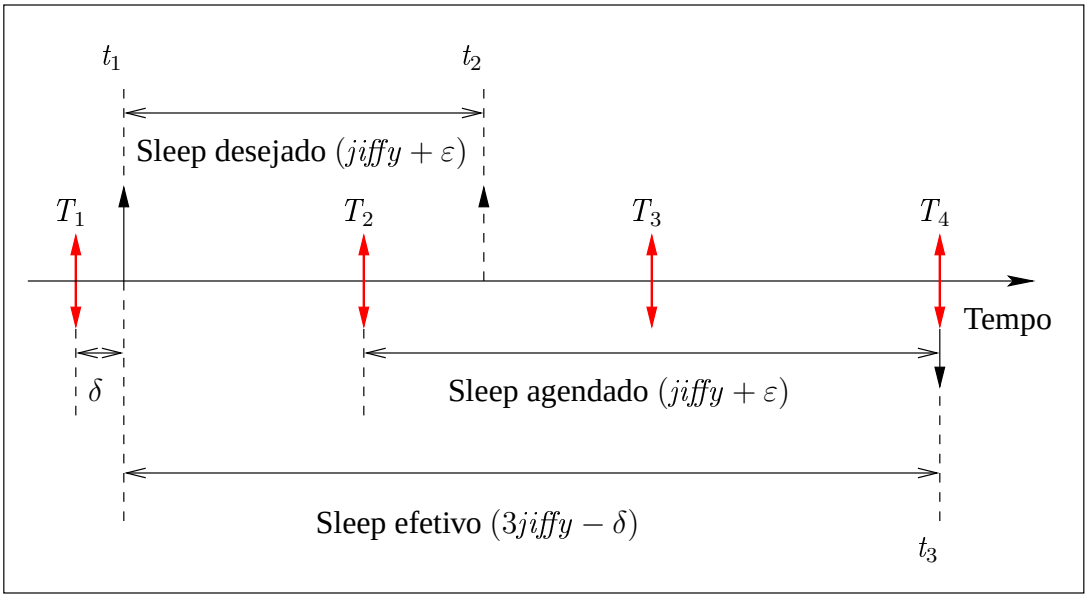


Figura 1.1: Latência causadas pela existência do *tick*

Para ilustrar o efeito da granularidade do *tick* na precisão de escalonamento (*scheduling jitter*), montamos um experimento com o *kernel* 2.6.19.7 com preempção do *kernel* habilitada (`CONFIG_PREEMPT`). O *tick* utilizado foi o valor padrão do kernel, $jiffy = 4ms$, correspondendo a uma freqüência de $250Hz$. Para o experimento, um processo executou sozinho e com a prioridade máxima em modo *single*, ou seja, com a carga mínima possível no sistema. Este processo foi programado para executar as operações seguintes, apresentadas sob forma de pseudo-código:

```
bef := read_tsc
while(1) {
     usleep(jiffy + ε )
     now := read_tsc
     print(now - bef)
     bef := now
}
```

Onde `read_tsc` é uma função que lê o valor do *Time Stamp Counter* (TSC), `bef` e `now` são duas variáveis que armazem o valor lido e $\varepsilon = 2ms$.

O resultado ótimo esperado para os valores das diferenças, se não fosse a existência do *tick*, seria de $6ms$. A figura 4.2 apresenta o tempo realmente observado entre duas chamadas sucessivas da chamada de sistema `usleep`. As duas execuções mostradas correspondem aos dois cenários de execução diferentes observados entre várias realizações deste experimento. Após o primeiro laço, os valores permanecem sempre iguais, portando, só foram mostrados os resultados obtidos para os 5 primeiros laços.

Observa-se que, nas duas execuções, depois da primeira chamada, os processos sempre dormem $8ms$, apesar do pedido de $6ms$ passado para a chamada `usleep`. Isto é uma conseqüência direta do valor do $jiffy$ de $4ms$, pois $8ms$ é o menor múltiplo do $jiffy$ maior que $6ms$. Observa-se também que na primeira chamada da primeira execução, o processo chega a dormir $11ms$ enquanto que na segunda execução, o processo dorme aproximadamente $7ms$. Este dois cenários diferentes ilustram a dependência do tempo de dormência no instante relativo no qual a primeira chamada `usleep` é efetuada em relação ao instante no qual o *tick* ocorre, conforme explicado no início desta seção.

É importante observar que a variabilidade discutida nesta seção caracteriza o comportamento do escalonador do Linux. No contexto deste trabalho, esta variabilidade não tem rele-

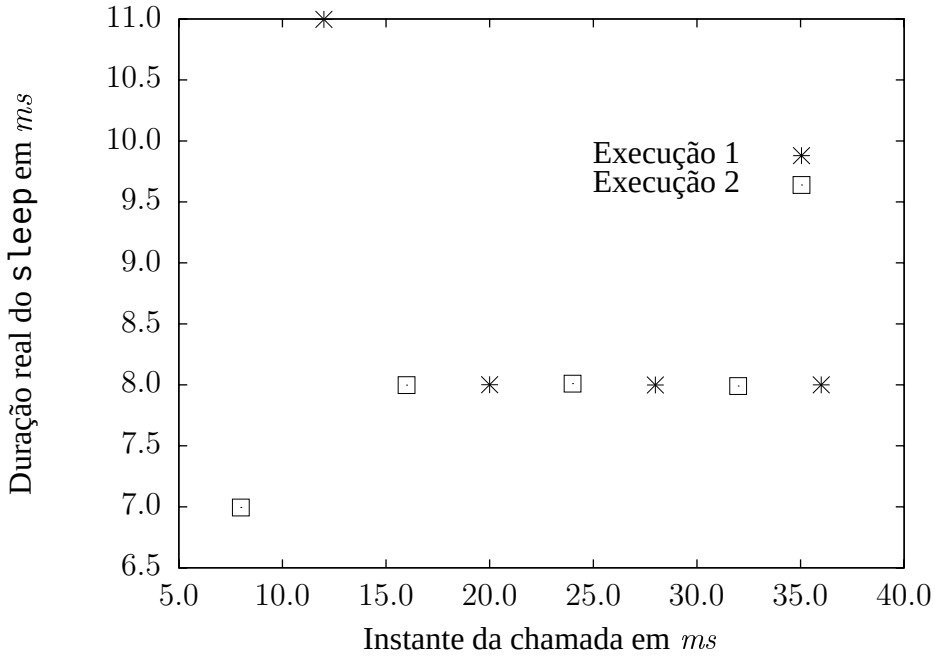


Figura 1.2: Tempo de dormência com chamadas `usleep` de 6 ms

vância, pois as plataformas de tempo real estudas utilizam um escalonador próprio, baseados em temporizadores de alta precisão. Portanto, a variabilidade de escalonamento não foi contemplada nos experimentos, pois não serve para efeito de comparação. No entanto, escolheu-se apresentar esta conseqüência da existência do *tick* devido à sua importância na implementação dos SO modernos.

1.3.2 Latência de interrupção

Como foi visto na seção 4.2.2, as interrupções de hardware são assíncronas e podem acontecer em qualquer momento do ciclo de execução do processador. Além disso, a execução da parte crítica de um tratador de interrupção requer eventualmente a desabilitação das interrupções para impedir o acesso concorrente a dados protegidos por *spin-locks* dentro de tratadores de interrupção.

Portanto, quando uma interrupção acontece, vários cenários de latência para a sua detecção e seu tratamento pelo processador são possíveis. Se as interrupções forem habilitadas, ela é detectada no final do ciclo das instruções em execução. Senão, a interrupção pode acontecer

enquanto as interrupções estão desabilitadas pela execução da parte crítica de um tratador de interrupção. Após o fim da execução deste tratador, as interrupções voltam a ser habilitadas novamente.

Em seguida à detecção da interrupção, o processador começa por gravar o contador de programa e alguns registradores da memória para poder retomar a execução do processo interrompido, depois do tratamento da interrupção. Observe que um tratador de interrupção executa no contexto do último processo executado. Conseqüentemente, a troca de contexto necessária para executar o tratador no processador é bastante rápida. Depois de executar mais algumas operações, tal como ler o vetor de interrupção no controlador de interrupção e carregar as informações necessárias no seus registradores, o processador finalmente começa a executar o tratador da interrupção que aconteceu. O tempo decorrido entre o instante no qual a interrupção aconteceu e o início da execução do tratador associado é chamado de **latência de interrupção** (*interrupt latency*).

A latência de interrupção caracteriza a capacidade do sistema para reagir a eventos externos. Portanto, esta grandeza foi contemplada como métrica para efeito de comparação das plataformas estudas (ver seção **??**).

1.3.3 Latência de ativação

Quando uma interrupção ocorre, quer seja porque um temporizador expirou ou porque um evento de hardware ocorreu, o tratador da interrupção executa imediatamente a parte crítica. Lembrar que a palavra crítica faz aqui referência às primitivas de sincronização e as estruturas de dados utilizadas, no contexto de um sistema preemptível e com acesso concorrente aos dados (ver seção 4.2.2). A parte não-crítica do tratador é executada num *softirq* logo após o retorno da parte crítica do tratador. No entanto, entre o instante no qual a interrupção ocorre e o instante no qual o *softirq* começa a executar, outras interrupções podem acontecer, provocando um possível atraso na execução da parte não-crítica.

Nas plataformas de tempo real, eventos de temporizadores ou de *hardware* são utilizados para disparar tarefas, num modelo similar aos *softirqs*. Tal tarefa, muitas vezes periódica, tem um contexto próprio e fica suspensa, na espera de um evento. Quando o evento ocorre, a interrupção associada aciona o seu tratador que, por sua vez, acorda a tarefa. O intervalo de

tempo entre os instantes no qual o evento ocorre e o início da execução da tarefa associada é chamada de **latência de ativação**. Assim como no caso dos *softirqs*, a latência de ativação pode ser aumentada pela ocorrência de interrupções. Além disso, a execução de outros *softirqs* pode ser escalonada com alguma política (ex: FIFO, prioridade fixa), o que pode também gerar interferências na latência de ativação.

Um outro aspecto importante diz respeito à implementação dos temporizadores utilizados para agendar as tarefas de tempo real. Mais especificamente, a obtenção de uma precisão de micro-segundos nos eventos disparados por temporizadores requer a utilização de relógios de alta precisão, distintos daqueles usados pelo *kernel* padrão. Desta forma, a latência de ativação passa a ser independente do escalonador de processos do Linux e do valor do `jiffy`.

Assim como a latência de interrupção, a latência de ativação caracteriza a capacidade de um sistema em reagir aos eventos externos. Portanto, esta grandeza foi também contemplada com métrica para efeito de comparação das plataformas estudas (ver seção **??**).

1.3.4 Latência causada pela inversão de prioridade

Para organizar o compartilhamento dos recursos, os processos utilizam as primitivas de *locks* que foram descritas na seção 4.2.4.1. Quando um processo quer obter o acesso a um recurso, ele adquire o *lock* associado. Uma vez em posse do *lock*, um processo utiliza o recurso e, em algum momento futuro, devolve o *lock* quando ela não precisa mais do recurso.

Este mecanismo de reserva pode causar latência de ativação em contradição com as políticas de prioridades definida no sistema. Considere-se, primeiramente, um cenário envolvendo dois processos P_A e P_B, o primeiro de alta prioridade e o segundo de baixa prioridade. Suponha que P_B adquire um recurso compartilhado R, e que, enquanto P_B está utilizando R, uma interrupção de *hardware* acorda P_A. Então, P_A causa a preempção de P_B e adquire o processador. Possivelmente, P_A tenta adquirir o *lock* do recurso R. Porém, P_B não liberou o *lock* ainda e, conseqüentemente, P_A tem que esperar que P_B ganhe o processador novamente e complete a sua execução, pelo menos até devolver o *lock*. Só então P_A pode adquirir o processador e conseguir o recurso R. Percebe-se que, neste cenário, o processo de mais alta prioridade acaba esperando pelo processo de mais baixa prioridade.

Esta situação simples pode se tornar ainda pior no seguinte caso. Suponha agora que um

(ou mais) processo P_M de prioridade média, mais alta que a prioridade de P_B e mais baixa que a prioridade de P_A comece a executar enquanto P_B está em posse do recurso R. P_A não pode executar enquanto P_B não libera o recurso R e P_B não pode executar enquanto P_M não libera o processador. Portanto, o processo de mais alta prioridade pode ficar esperando um tempo indefinido, enquanto tiver processos de prioridade intermediária ocupando o processador.

Duas soluções para este problema, chamado de inversão de prioridade, foram propostas por Sha et al em 1990 num trabalho pioneiro (SHA; RAJKUMAR; LEHOCZKY, 1990). A primeira utiliza o conceito de herança de prioridade. Resumidamente, enquanto um processo de baixa prioridade utiliza um recurso, ele herda a prioridade do processo de maior prioridade que está esperando por aquele recurso. Desta forma, e na ausência de bloqueios encadeados, o tempo máximo que um processo de alta prioridade terá de esperar é o tempo máximo que um processo de mais baixa prioridade pode bloquear o recurso. A segunda solução consiste em determinar em tempo de projeto, a mais alta prioridade dos processos que compartilham um certo recurso. Esta prioridade teto será então atribuída a qualquer um destes processos durante sua utilização deste recurso. Apesar de não impedir a inversão de prioridade, estas soluções permitem limitar o tempo máximo de espera de um processo de mais alta prioridade no sistema, aumentando, portanto, o grau de previsibilidade do sistema. Outras soluções são baseadas em protocolos sem *locks* ou na replicação dos recursos (TANENBAUM, 2001).

A ocorrência de inversão de prioridade depende altamente das aplicações e do uso que elas fazem dos recursos compartilhados. Além disso, as latências por inversão de prioridade sofridas pelas aplicações resultam também das latências de interrupção e de ativação. Ambos os fatos dificultam a elaboração de experimentos de medição. Portanto, esta grandeza não foi utilizada como métrica no contexto deste capítulo.

1.4 SOLUÇÕES DE SOTR BASEADAS EM LINUX

Nesta seção, apresentaremos os princípios de três abordagens diferentes para tornar o Sistema Operacional Linux de Tempo Real. A primeira, descrita na seção 4.4.1, consiste em tornar o *kernel* Linux inteiramente preemptível. Desta forma, é possível limitar as latências máximas de interrupção e de ativação. Uma outra abordagem, descrita na seção 4.4.2, consiste em organizar a ativação das tarefas de tempo real a partir dos tratadores de interrupção, criando uma

interface de programação acessível em modo usuário. Finalmente, uma terceira abordagem, baseada em *nanokernel*, será descrita na seção 4.4.3. Esta proposta utiliza uma camada intermediária entre o *kernel* e o hardware, o *nanokernel*, que oferece serviços de tempo real para as aplicações. Apesar de existir outras propostas de SOTR baseadas no Linux (ex: (BARABANOV, 1997; SRINIVASAN et al., 1998; CALANDRINO et al., 2006)), estas três abordagens são bastante representativas para permitir a exposição dos princípios fundamentais destinados ao aumento da previsibilidade de um SOPG tal como Linux.

1.4.1 O *patch* PREEMPT-RT

Há alguns anos, Ingo Molnar, juntamente com outros desenvolvedores do *kernel* Linux (MCKENNEY, 2005; ROSTEDT; HART, 2007), têm trabalhando ativamente para que o próprio *kernel* possa oferecer serviços de tempo real confiáveis. Este trabalho resultou na publicação do *patch* do *kernel* chamado "PREEMPT-RT" em abril de 2006 (I. Molnar et al., 2008).

Para resolver o problema da precisão temporal do escalonador baseado em *ticks* descrito na seção 4.3.1, o *patch* PREEMPT-RT utiliza uma nova implementação dos temporizadores de alta resolução desenvolvida por Thomas Gleixner (L. Torvalds et al., 2008). Baseados no registrador *Time Stamp Counter* (TSC) da arquitetura Intel ou em relógios de alta resolução, esta implementação oferece uma API que permite obter valores temporais com uma resolução de micro-segundos. Desde a versão do *kernel* 2.6.21, esta API faz parte da linha principal do *kernel*. De acordo com resultados apresentados (ROSTEDT; HART, 2007), os tempos de latência de ativação obtidos usando esta API são da ordem de algumas dezenas de micro-segundos e não dependem mais da freqüência do *tick*. Com PREEMPT-RT, o *tick* continua sendo estritamente periódico. No entanto, com a proposta do KURT-Linux (SRINIVASAN et al., 1998), que disponibiliza o *patch* $UTIME$, é possível utilizar um *tick* aperiódico e programável com uma resolução de alguns micro-segundos.

O segundo problema diz respeito aos tempos de latência de interrupção e de preempção. Além de utilizar temporizadores de alta precisão, o *patch* PREEMPT-RT comporta várias modificações para tornar o *kernel* totalmente preemptível. Este objetivo é essencial para garantir que quando um processo de mais alta prioridade acorda, ele consegue adquirir o processador com uma latência mínima, sem ter que esperar o fim da execução de um processo de menor pri-

oridade, mesmo que este esteja executando em modo *kernel*. Como foi visto nas seções 4.2.4.1 e 4.3.2, a utilização das primitivas de sincronização que permitem garantir a exclusão mútua de regiões críticas introduz possíveis fontes de latência e indeterminismo. Para eliminar estas fontes de imprevisibilidade, PREEMPT-RT modifica estas primitivas de maneira a permitir a implementação de um protocolo complexo, baseado em herança de prioridade. Por exemplo, um *spin-lock* (ou um *mutex*) agora possui um dono, uma fila de processos em espera e pode sofrer preempção, ou seja, um processo possuindo um *spin-lock* pode ser suspenso. Os atributos dono e fila são necessários para a implementação do protocolo baseado em herança de prioridade (SHA; RAJKUMAR; LEHOCZKY, 1990), como foi visto na seção 4.3.

Uma outra modificação importante diz respeito ao tratamento das interrupções. No *kernel* padrão, quando uma interrupção acontece, a parte crítica do tratador da interrupção é executada logo que a interrupção é detectada pelo processador, podendo, conseqüentemente, atrasar a execução de um outro tratador ou processo de maior prioridade. Para diminuir esta causa de latência, o *patch* PREEMPT-RT utiliza *threads* de interrupções. Quando uma linha de interrupção é inicializada, um *thread* é criado para gerar as interrupções associadas a esta linha. Na ocorrência de uma interrupção, o tratador associado simplesmente mascara a interrupção, acorda o *thread* da interrupção e volta para o código interrompido. Desta forma, a parte crítica do tratador de interrupção é reduzida ao seu mínimo e a latência causada pela sua execução, além de ser breve, é determinística. O *thread* acordado será eventualmente escalonado, de acordo com a sua prioridade e os demais processos em execução no processador.

De acordo com os resultados obtidos (ROSTEDT; HART, 2007; SIRO; EMDE; MCGUIRE, 2007), o *patch* PREEMPT-RT permite reduzir as latências do *kernel* padrão para valores da ordem de algumas dezenas de micro-segundos. Portanto, usando códigos confiáveis, respeitando as regras de programação do *patch* e alocando os recursos de acordo com os requisitos temporais, a solução PREEMPT-RT tem a vantagem de oferecer o ambiente de programação do sistema Linux, dando acesso às bibliotecas C e ao conjunto de software disponível para este sistema.

1.4.2 Uma caixa de areia em espaço usuário

A proposta de caixa de areia (QI; PARMER; WEST, 2004; FRY; WEST, 2007), tem o objetivo de permitir a extensão do *kernel* Linux padrão para oferecer uma interface de programação

para tarefas de tempo real executadas em modo usuário. A idéia fundamental aplicada aqui é criar uma extensão do espaço de memória de todos os processos executados no sistema. Esta implementação utiliza o gerenciamento por páginas da memória virtual e não requer nenhum dispositivo de hardware específico. Basicamente, uma caixa de areia corresponde a uma ou duas páginas da memória virtual que são adicionadas a todas as áreas de memória dos processos do sistema. Desta forma, qualquer código da caixa de areia pode ser executado em modo usuário no contexto de qualquer processo.

Para oferecer as extensões da caixa de areia, o *kernel* é modificado através de módulos carregados, cujas as funcionalidades são utilizadas via *ioctls*, de uma forma semelhante aos controladores de dispositivos. Através da interface de programação, um processo P pode registrar serviços na caixa de areia. Para utilizar estes serviços, por exemplo, durante a execução de um tratador de interrupção, o *kernel* usa uma função de *upcall* que acorda um *thread* previamente criado pelo processo P. Este *thread* executa em modo usuário, no contexto do último processo que estava executando no processador. Como este processo, possivelmente diferente de P, tem a caixa de areia na sua área de memória virtual, o *thread* acordado tem acesso a todas as funcionalidades registradas nesta área de memória compartilhada.

O tamanho da caixa de areia é arbitrário, mas deve ser suficiente para comportar uma pilha de execução dos *thread* e para conter uma versão pequena da biblioteca C padrão. Para isto, duas páginas de $4Mb$ são utilizadas. Uma página, chamada "pública", dá direito de leitura e execução, tanto em modo usuário quanto em modo *kernel*. Esta página contém as funções da interface de programação e as funções registradas pelos processos quando eles são criados. A outra página, dita "protegida", dá direito de leitura e escrita em modo *kernel*, mas só pode ser escrita por um *thread* executando em modo usuário durante um *upcall*. Isto garante que os dados de um processo, contidos na caixa de areia, não podem ser alterados por outros processos.

Nesta implementação, os serviços da caixa de areia são obtidos a partir da parte não crítica do tratador de interrupção (*softirq*). Apesar desta parte dos tratadores ser executada com certa imprevisibilidade comparativamente com a parte crítica, esta escolha é justificada pelos autores pelo fato de permitir aos códigos da caixa de areia fazer chamadas de sistemas bloqueantes. Resultados apresentados mostram que os tempos de latência de interrupção são da ordem de dezenas de micro-segundos, inclusive no caso de interrupções geradas pela placa de rede na recepção de mensagens Ethernet. No entanto, no caso de eventos de redes, os autores produzem

desvios padrões da mesma ordem de grandeza que as latências medidas. Apesar deste fato, o uso da caixa de areia é advogado pelos autores (FRY; WEST, 2007) para sistemas de tempo real não-críticos. No contexto deste trabalho, esta solução não foi contemplada, pois *DoRiS* tem requisitos de tempo real críticos.

1.4.3 *Nanokernel*

As soluções de implementação para SOTRs baseadas em *nanokernel*, sendo as mais divulgadas RT-Linux (BARABANOV, 1997; V. Yodaiken et al., 2008), *Real Time Application Interface* (RTAI) (DOZIO; MANTEGAZZA, 2003; P. Mantegazza et al., 2008) e Xenomai (GERUM, 2005; P. Gerum et al., 2008) são as únicas, até o momento, que alcançam latências da ordem do micro-segundos e dão suporte a sistemas críticos. Estas soluções utilizam uma camada de indireção das interrupções, chamada camada de abstração do hardware (HAL), localizada entre o *kernel* e os dispositivos de hardware e disponibilizam uma interface de programação para serviços de tempo real. Observa-se que os códigos fontes do Xenomai e RTAI são disponíveis sob licença GNU/GPL. No caso do RT-Linux, uma versão profissional é desenvolvida sob licença comercial. Uma outra versão livre é disponibilizada, sob licenças específicas, com uma interface de utilização restrita e sem suporte para as versões do *kernel* posteriores a 2.6.9.

Do ponto de vista da implementação, ambos Adeos e o *nanokernel* do RT-Linux utilizam o mecanismo de virtualização das interrupções, também chamada de indireção de interrupção, introduzido na técnica de “proteção otimista das interrupções” (STODOLSKY; CHEN; BERSHAD, 1993). No contexto da interação do *nanokernel* com Linux, esta técnica pode ser resumida da seguinte maneira. Quando uma interrupção acontece, o *nanokernel* identifica se esta é relativa a uma tarefa de tempo real ou se a interrupção é destinada a um processo do *kernel* Linux. No primeiro caso, o tratador da interrupção é executado imediatamente. Caso contrário, a interrupção é enfileirada e, em algum momento futuro, entregue para o *kernel* Linux quando nenhuma tarefa de tempo real estiver precisando executar. Quando o *kernel* Linux precisa desabilitar as interrupções, o *nanokernel* deixa o *kernel* Linux acreditar que as interrupções estão desabilitadas. No entanto, o *nanokernel* continua a interceptar qualquer interrupção de *hardware*. Nesta ocorrência, a interrupção é tratada imediatamente se for destinada a uma tarefa de tempo real. Caso contrário, a interrupção é enfileirada, até que o *kernel* Linux habilite suas

interrupções novamente.

1.4.3.1 RT-Linux No caso da versão livre do RT-Linux, tanto a camada HAL quanto API de programação são fornecidas em um único *patch* que modifica o código fonte do *kernel* Linux. Este *patch* permite então co-existência do *nanokernel* RT-Linux que oferece garantias temporais críticas para as tarefas de tempo real e do *kernel* Linux padrão. O modelo de programação das tarefas de tempo real é baseado na inserção de módulos no *kernel* em tempo de execução. Portanto, estas tarefas devem necessariamente executar em modo protegido, o que restringe a interface de programação fornecida pelo RT-Linux.

Para realizar a virtualização das interrupções, as funções que manipulam as interrupções `local_irq_enable`, antigamente `sti` (*set interruption*) e `local_irq_disable`, antigamente `cli` (*clear interruption*), são modificadas. Para dar o controle à camada HAL, as novas funções não alteram mais a máscara real de interrupção, mas utilizam uma máscara virtual. Enquanto esta máscara virtual indicar que as interrupções são desabilitadas, depois de uma chamada `local_irq_disable` pelo *kernel* Linux, o *nanokernel* intercepta as interrupções que não são de tempo real e trata as interrupções de tempo real imediatamente. Quando o *kernel* chama a função `local_irq_disable`, a máscara virtual de interrupção é modificada e o *nanokernel* entrega as interrupções pendentes para Linux.

Resumindo, no RT-Linux, o *kernel* Linux é uma tarefa escalonada pelo *nanokernel* RT-Linux como se fosse a tarefa de mais baixa prioridade no sistema.

1.4.3.2 Adeos, RTAI e Xenomai No caso dos projetos RTAI (DOZIO; MANTEGAZZA, 2003) e Xenomai (GERUM, 2005), a camada HAL é fornecida pelo *Adaptative Domain Environment for Operating Systems* (Adeos), desenvolvida por Karim Yaghmour (YAGHMOUR, 2001). Adeos, fornecida por um *patch* separado, tem os seguintes objetivos:

- Permitir o compartilhamento dos recursos de hardware entre diferentes sistemas operacionais e/ou aplicações específicas.
- Contornar a padronização dos SO, isto é, flexibilizar o uso do hardware para devolver o controle aos desenvolvedores e administradores de sistema.

- Oferecer uma interface de programação simples e independente da arquitetura.

Para não ter que iniciar a construção de um sistema operacional de tempo real completo, o *nanokernel* Adeos utiliza Linux como hospedeiro para iniciar o hardware. Logo no início, a camada Adeos é inserida abaixo do *kernel* Linux para tomar o controle do hardware. Após isto, os serviços de Adeos podem ser utilizados por outros sistemas operacionais e/ou aplicações executando conjuntamente ao *kernel* Linux.

A arquitetura Adeos utiliza dois conceitos principais, domínio e canal hierárquico de interrupção. Os domínios correspondem aos diferentes códigos hospedados no sistema. Um domínio pode tanto corresponder a um sistema operacional completo como Linux quanto a uma aplicação mais específica como RTAI ou Xenomai. Um domínio enxerga Adeos mas não enxerga os outros domínios hospedados no sistema.

O canal hierárquico de interrupção, chamado *ipipe*, serve para priorizar a entrega da interrupções entre os domínios. Quando um domínio se registra no Adeos, ele é colocado numa posição no *ipipe* de acordo com os seus requisitos temporais. Adeos utiliza então o mecanismo de virtualização das interrupções para organizar a entrega hierárquica das interrupções, começando pelo domínio mais prioritário e seguindo com os menos prioritários. Funções apropriadas (*stall/unstall*) permitem bloquear ou desbloquear a transmissão das interrupções através de cada domínio.

Para prover serviços de tempo real, as interfaces RTAI ou Xenomai utilizam o domínio mais prioritário do *ipipe*, chamado “domínio primário”. Este domínio corresponde, portanto, ao núcleo de tempo real no qual as tarefas são executadas em modo protegido. Como as interrupções são entregues começando pelo domínio primário, o núcleo de tempo real pode escolher atrasar, ou não, a entrega das interrupções para os demais domínios registrados no *ipipe*, garantindo, desta forma, a execução das suas próprias tarefas. Módulos podem ser utilizados para carregar as tarefas, de forma semelhante ao RT-Linux.

Nas plataformas Xenomai e RTAI, o “domínio secundário” corresponde ao *kernel* Linux. Neste domínio, o conjunto de bibliotecas e software usual do Linux está disponível. Em contrapartida, as garantias temporais são mais fracas, dado que o código pode utilizar as chamadas de sistemas bloqueantes do Linux.

Para oferecer o serviço de tempo real em modo usuário chamado LXRT, RTAI utiliza o mecanismo de associação (*shadowing*) de um *thread* do núcleo de tempo real com um processo usuário executando no domínio Linux. De acordo com sua prioridade, este *thread*, também chamado de "sombra" do processo, executa o escalonamento rápido do seu processo associado.

O projeto Xenomai se distingue do RTAI/LXRT. Além de não utilizar o mecanismo de associação (*shadowing*), Xenomai tem por objetivo privilegiar o modelo de programação em modo usuário. O modelo de tarefas executando no modo protegido só está sendo mantido para dar suporte às aplicações legadas. A implementação dos serviços de tempo real em modo usuário se baseia nas seguintes regras:

- A política de prioridade utilizada para as tarefas de tempo real, e para o escalonador associado, é comum aos dois domínios.
- As interrupções de hardware não podem impedir a execução de uma tarefa prioritária enquanto ela estiver no domínio secundário.

A primeira destas garantias utiliza um mecanismo de herança de prioridade entre o domínio primário e o domínio secundário. Quando uma tarefa do domínio primário migra para o secundário, por exemplo, porque ela efetua uma chamada de sistema do Linux, esta tarefa continua com a mesma prioridade, maior que a de qualquer processo do Linux. Este mecanismo garante notadamente que a preempção de uma tarefa de tempo real executando no domínio secundário não possa ser causada por uma tarefa de menor prioridade executando no domínio primário. A segunda garantia é obtida por meio de um domínio intermediário, chamado de "escudo de interrupção", registrado no *ipipe*, entre o primeiro domínio (o *nanokernel*) e o segundo domínio (o *kernel* Linux). Enquanto uma tarefa de tempo real está executando no segundo domínio, o "escudo de interrupção" é utilizado para bloquear as interrupções de hardware destinadas ao Linux. No entanto, aquelas interrupções destinadas à tarefa em execução são entregues imediatamente.

Para completar esta arquitetura, as chamadas de sistema executadas por uma tarefa enquanto está no domínio primário e secundário são interceptadas por Adeos que determina a função a ser executada, seja ela do Linux padrão ou da API do Xenomai. Isto permite que uma tarefa executando no domínio secundário possa obter os serviços providos por Xenomai.

Observa-se que quando uma tarefa está no domínio secundário, ela pode sofrer uma latência de escalonamento devido à execução de alguma sessão não preemptiva do Linux. Portanto, Xenomai se beneficia do esforço de desenvolvimento do *patch* PREEMPT-RT, que tem por objetivo tornar o *kernel* inteiramente preemptível. Apesar de ainda constituir um *patch* separado, várias propostas do grupo de desenvolvedores deste *patch* já foram integrados na linha principal do *kernel* 2.6, melhorando significativamente suas capacidades de preempção.

1.5 METODOLOGIA EXPERIMENTAL

Em geral, realizar medições precisas de tempo no nível dos tratadores de interrupção pode não ser tão simples. De fato, o instante exato no qual uma interrupção acontece é de difícil medição, pois tal evento é assíncrono e pode ser causado por qualquer dispositivo de *hardware*. Para obter medidas das latências de interrupção e ativação confiáveis com alto grau de precisão, aparelhos externos, tais como osciloscópios ou outros computadores, são necessários. No entanto, como o objetivo do presente trabalho não foi medir estas latências de forma precisa, mas caracterizar e comparar o grau de determinismo das plataformas operacionais estudadas, adotou-se uma metodologia experimental simples e efetiva, que pode ser reproduzida facilmente em outros contextos.

1.5.1 Configuração do experimento

O dispositivo experimental utilizou três estações: (1) a estação de medição E_M, na qual os dados foram coletados e onde temos uma tarefa de tempo real τ à espera de eventos externos; (2) a estação de disparo E_D, que foi utilizada para enviar pacotes Ethernet com uma freqüência fixa à estação E_M; e (3) a estação de carga E_C, utilizada para criar uma carga de interrupção na estação E_M. As estações de disparo e carga foram conectadas a estação de medição por duas redes Ethernet distintas, conforme ilustrado no diagrama da figura 4.3.

As chegadas em E_M dos pacotes enviados por E_D servem para disparar uma cascata de eventos na estação E_M, permitindo a simulação de eventos externos via a porta paralela (PP). Mais explicitamente, cada chegada de um pacote Ethernet na placa foi utilizada para disparar uma interrupção na PP, escrevendo no pino de interrupção desta porta. Esta escrita foi realizada

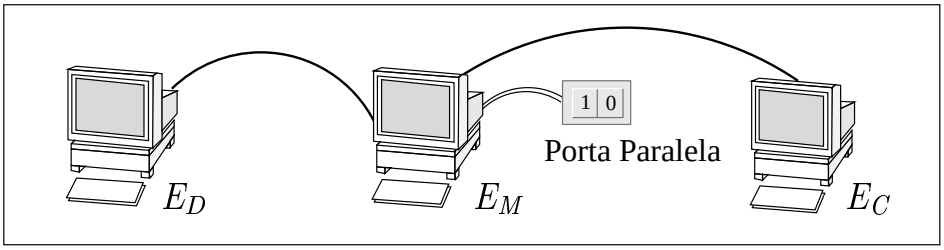


Figura 1.3: Configuração do experimento.

pelo próprio tratador T_{eth} de interrupção da placa de rede de E_M. O instante t_1 de escrita no pino de interrupção da PP pelo tratador T_{eth} constitui então o início da seqüência de eventos utilizados para medir as latências de interrupção (Lat_{irq}) e de ativação (Lat_{ativ}).

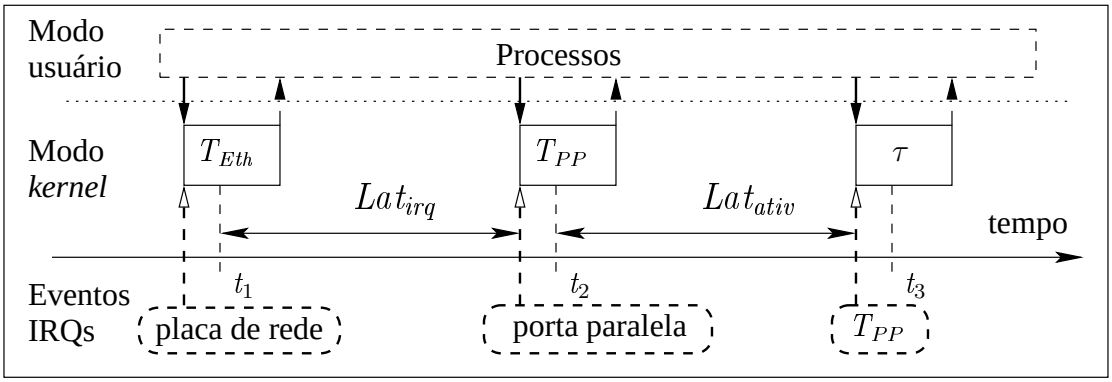


Figura 1.4: Cálculo das latências de interrupção e ativação na estação E_M.

As medidas de latência foram realizadas pela consulta do contador do processador, chamado *Time Stamp Counter* (TSC), permitindo uma precisão da ordem de micro-segundos. Concretamente, um módulo específico (M) foi desenvolvido na estação E_M para exportar as funcionalidades seguintes:

- Ler os 64 bits do TSC e armazenar o valor lido na memória.
- Escrever na porta de entrada e saída 0x378 da porta paralela para gerar interrupções de hardware.
- Criar uma tarefa τ que, num sistema real, executaria algum código útil precisando de garantias de tempo real crítico. Neste experimento, τ simplesmente grava o valor do TSC.

- Requisitar a captura das interrupções na linha 7 associada à porta paralela e registrar o tratador de interrupção T_{PP} associado.
- Definir o tratador T_{PP} que executa as duas operações seguintes: (1) gravar o TSC; (2) acordar tarefa τ.
- Criar um canal de comunicação FIFO assíncrono para a transferência dos dados temporais coletados em modo protegido para o espaço usuário.

Parte deste conjunto de funções foi disponibilizado sob forma de `ioctl` para os processos executando em modo usuário através de um arquivo especial `/dev/M` do sistema de arquivos. A outra parte foi diretamente exportada, sob a forma de serviços do *kernel* podendo ser utilizados somente por *threads* do *kernel*.

As medidas seguiram o seguinte roteiro, ilustrado pela figura 4.4:

- A estação E_D envia pacotes Ethernet para a estação E_M, provocando interrupções assíncronas em relação as aplicações executando em E_M.
- A interrupção associada à chegada de um pacote provoca a preempção da aplicação executando pelo tratador de interrupção T_{eth}.
- T_{eth} escreve no pino de interrupção da PP e o instante t_1 é armazenado na memória. Este valor t_1 corresponde ao valor lido no relógio local, no instante na escrita da PP, logo após a chegada de uma pacote Ethernet.
- A interrupção associada à escrita no pino de interrupção da PP provoca a preempção da aplicação executando pelo tratador de interrupção T_{PP}.
- T_{PP} grava o instante t_2 e acorda a tarefa τ. Este valor t_2 corresponde ao valor do relógio local, logo após o início de T_{PP}.
- No momento que a tarefa τ acorda, ela grava o instante t_3 e volta a ficar suspensa até a próxima interrupção na PP. O valor de t_3 corresponde ao tempo no qual a tarefa τ começa a executar, no final da cascata de eventos provocada pela chegada de um pacote na placa de rede.

Assim como representado na figura 4.4, Lat_{irq} corresponde a diferença $t_2 - t_1$ e Lat_{ativ} a diferença $t_3 - t_2$.

No decorrer do experimento, a transferência das medições da memória para o sistema de arquivos é efetuada por um processo usuário (P) (não representado na figura). Antes de iniciar a fase de medidas, P abre o arquivo especial */dev/M* para poder ter acesso às funções *ioctl* disponibilizadas pelo módulo M. P entra então num laço infinito no qual ele executa a chamada de sistema `read` para ler os dados do canal FIFO. Enquanto não há dados, P fica bloqueado. Em algum momento depois da escrita de dados pela tarefa τ, P acorda, lê o canal e escreve os dados num arquivo do disco rígido. Tal procedimento permitiu impedir qualquer interferência entre a aquisição dos dados e seu armazenamento no sistema de arquivos, pois a execução do processo P em modo usuário, não interfere na execução dos módulos do experimento, executados em modo *kernel*.

O mecanismo do disparo das interrupções na porta paralela pelos eventos de chegada de pacotes na placa de rede foi utilizado para garantir que estas interrupções ocorressem de forma assíncrona em relação aos processos executando na estação de medição. É interessante observar que a interrupção na porta paralela sempre acontece logo após a execução de T_{Eth}. Portanto, a medida da latência de interrupção Lat_{irq} deve ser considerada apenas como indicativa. No entanto, ela pôde ser utilizada para o objetivo principal destes experimentos, isto é, comparar as diferentes plataformas estudadas.

Como foi visto na seção 4.3.3, a latência de ativação Lat_{ativ} caracteriza o tempo necessário para ativar uma tarefa após a ocorrência do seu evento de disparo. O dispositivo apresentado aqui permitiu obter medidas quantitativamente corretas desta latência. Além disso, o efeito do aumento da atividade na estação de medição pôde ser investigado, pois o instante de disparo da interrupção na porta paralela era totalmente independente da atividade dos processos na estação de medição.

1.5.2 Cargas de I/O, processamento e interrupção

Num primeiro momento, realizou-se experimentos com uma carga mínima no processador da estação de medição (modo *single*). Desta forma, observou-se o comportamento temporal das três plataformas em situação favorável. Em seguida, dois tipos de cargas foram utilizados si-

multaneamente para sobrecarregar a estação de medição. Tais sobrecargas tiveram por objetivo avaliar a capacidade de cada plataforma em garantir uma latência determinista no tratamento das interrupções e na ativação de tarefas de tempo real, apesar da existência de outras atividades não-críticas. As cargas de I/O e processamento foram realizadas executando as instruções seguintes:

```
while "true"; do
      dd if=/dev/hda2 of=/dev/null bs=1M count=1000
      find / -name ”*.c” | xargs egrep include
      tar -cjf /tmp/root.tbz2 /usr/src/linux-xenomai
      cd /usr/src/linux-preempt; make clean; make
done
```

Um outro estresse de interrupção foi criado utilizando uma comunicação UDP entre a estação E_M configurada como servidor e a estação E_C configurada como cliente. Para isolar esta comunicação da comunicação entre E_M e E_D, utilizou-se uma segunda placa de rede em E_M, assim com ilustrado pelo diagrama da figura 4.4. Durante o experimento, o cliente transmitiu pequenos pacotes de 64 *bytes* na freqüência máxima permitida pela rede, ou seja, com uma freqüência superior a $200kHz$ (um pacote a cada $10\mu s$). Desta forma, mais de 100.000 interrupções por segundos foram geradas pela segunda placa de rede de E_M. Esta placa de rede foi registrada na linha de interrupção 18 cuja prioridade é menor que a prioridade da porta paralela. Portanto, as interrupções geradas nesta placa de rede não deveriam, a princípio, interferir nas latências de interrupção mensuradas.

Nos experimentos com cargas, os dois tipos de estresses foram aplicados simultaneamente e as medições só foram iniciadas alguns segundos depois.

1.6 AVALIAÇÃO DE LINUX$^{\mathbf{PRT}}$ E LINUX$^{\mathbf{XEN}}$

1.6.1 Configuração

Os experimentos foram realizados em computadores Pentium 4 com processadores de 2.6 Ghz e 512 Mb de memória, com o objetivo de ilustrar o comportamento temporal das três plataformas seguintes:

- **Linux$^{\mathbf{Std}}$:** Linux padrão - *kernel* versão 2.6.23.9 (opção *low-latency*);
- **Linux$^{\mathbf{Prt}}$:** Linux com o *patch* PREEMPT-RT (rt12) - *kernel* versão 2.6.23.9.
- **Linux$^{\mathbf{Rtai}}$:** Linux com o *patch* RTAI - versão "magma" - *kernel* versão 2.6.19.7;
- **Linux$^{\mathbf{Xen}}$:** Linux com o *patch* Xenomai - versão 2.4-rc5 - *kernel* versão 2.6.19.7;

Utilizou-se a configuração Linux$^{\mathbf{Std}}$ para realizar experimentos de referência para efeito de comparação com as duas plataformas de tempo real Linux$^{\mathbf{Prt}}$ e Linux$^{\mathbf{Xen}}$. A versão estável do *kernel* 2.6.23.9, disponibilizada em dezembro de 2007 foi escolhida para o estudo de Linux$^{\mathbf{Prt}}$, pois este *patch* tem evoluído rapidamente desde sua primeira versão publicada há dois anos. No entanto, utilizou-se a versão do *kernel* 2.6.19.7 para o estudo das versões "magma" de RTAI e 2.4-rc5 de Xenomai. De fato, considerou-se desnecessário atualizar a versão do *kernel*, pois RTAI e Xenomai são baseadas no Adeos (ver seção **??**) e, portanto, as garantias temporais oferecidas para as aplicações executando no primeiro domínio dependem apenas da versão de RTAI ou Xenomai e do *patch* Adeos associado, e não, da versão do *kernel* Linux.

Utilizou-se uma freqüência de disparo dos eventos pela estação E_D de 20Hz. Para cada plataforma, dois experimentos de 10 minutos foram realizados. O primeiro sem carga nenhuma do sistema e o segundo aplicando os estresses apresentados na seção 4.5.2. Para a plataforma Linux$^{\mathbf{Xen}}$, o experimento com estresses foi repetido por uma duração de 12 horas.

Para as duas plataformas Linux$^{\mathbf{Rtai}}$ e Linux$^{\mathbf{Xeno}}$, os diferentes testes de latências fornecidos foram utilizados, dando resultados menores que $10\ \mu s$ no pior caso para a latência de interrupção, conforme os padrões da arquitetura Intel Pentium 4. Em ambas plataformas, desabilitou-se as interrupções de gerenciamento do sistema (*SMI*), conforme as recomendações dos desenvolvedores (P. Mantegazza et al., 2008; P. Gerum et al., 2008). Isto cancelou uma latência periódica (a cada 32 segundos) de aproximadamente $2.5ms$ que aparecia nos testes. No caso das plataforma Linux$^{\mathbf{Std}}$ e Linux$^{\mathbf{Prt}}$, estas interrupções foram também desabilitadas, porém, não foi constatado nenhuma alteração das medidas realizadas.

1.6.2 Resultados experimentais

Os resultados experimentais são apresentados nas figuras 4.5 e 4.6, onde o eixo horizontal representa o instante de observação variando de 0 a 60 segundos e o eixo vertical representa

as latências medidas em μs. Apesar de cada experimento ter durado no mínimo uma hora, escolheu-se apresentar apenas resultados para um intervalo de $60s$, pois este intervalo é suficiente para observar o padrão de comportamento de cada plataforma. Neste intervalo, o total de eventos por experimentos é 1200, pois a freqüência de chegada de pacotes utilizada foi de $20Hz$.

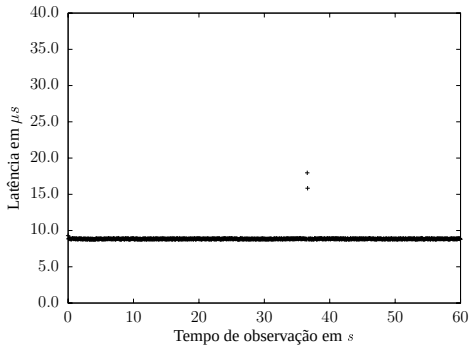
Abaixo de cada figura, os seguintes valores são indicados: Valor Médio (VM), desvio padrão (DP), valor mínimo (Min) e valor máximo (Max). Estes valores foram obtidos considerando a duração de uma hora de cada experimento. Na medida do possível, utilizou-se uma mesma escala vertical para todos os gráficos. Conseqüentemente, alguns valores altos podem ter ficado fora das figuras. Tal ocorrência foi representada por um triângulo próximo do valor máximo do eixo vertical.

1.6.2.1 Latência de interrupção A figura 4.5 apresenta as latências de interrupção medidas, com e sem estresse do sistema. Como pode ser observado, sem carga, o Linux^{Std}, Linux^{Rtai} e Linux^{Xen} tem comportamentos parecidos. Com carga, observa-se uma variação significativa do Linux^{Std}, como esperado.

Com relação ao Linux$^{\mathbf{Prt}}$, dois resultados chamam atenção. Primeiro, o comportamento do sistema sem carga exibe latências da ordem de 20 μs. Isto é causado pela implementação dos *threads* de interrupção vista na seção 4.4.1. Segundo, contradizendo as expectativas, a aplicação do estresses teve um impacto significativo, provocando uma alta variabilidade das latências. De fato, entre o instante no qual o tratador T_{PP} acorda o *thread* de interrupção e o instante no qual este *thread* acorda efetivamente, uma ou várias interrupções podem ocorrer. Neste caso, a execução dos tratadores associados pode provocar o atraso da execução de T_{PP}.

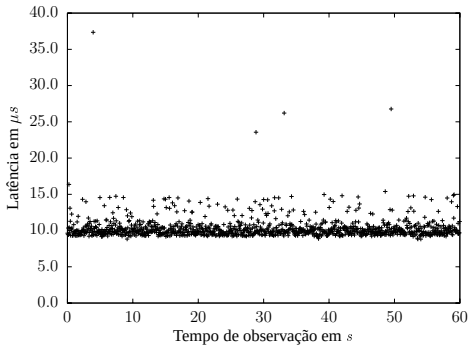
Para cancelar esta variabilidade indesejável, é possível usar o Linux$^{\mathbf{Prt}}$ sem utilizar a implementação de *threads* de interrupção. Para tanto, usa-se a opção `IRQF_NODELAY` na requisição da linha de interrupção. Utilizando esta opção na requisição da linha de interrupção da porta paralela, o comportamento do Linux$^{\mathbf{Prt}}$ passa a ser semelhante ao Linux$^{\mathbf{Std}}$.

1.6.2.2 Latência de ativação A figura 4.6 apresenta os resultados para as latências de ativação sem estresse e com estresse do processador. Como pode ser observado, o comportamento



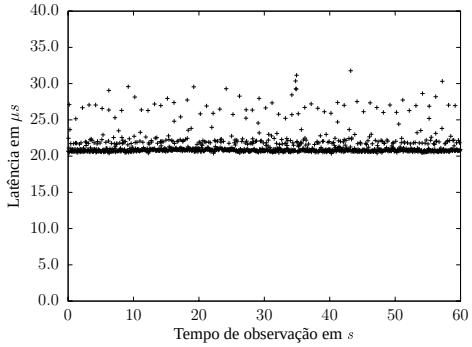
(a) **Linux$^{\mathbf{Std}}$ - Sem carga**

VM: 8.9, DP: 0.3, Min: 8.7, Max: 18.4



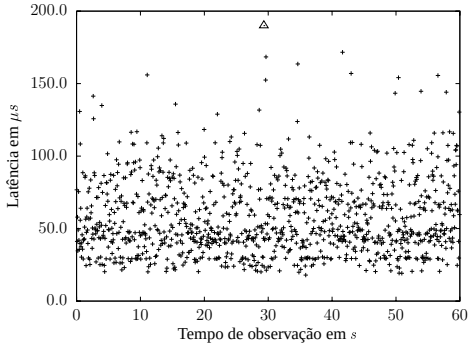
(b) **Linux$^{\mathbf{Std}}$ - Com carga**

VM: 10.4, DP: 1.9, Min: 8.8, Max: 67.7



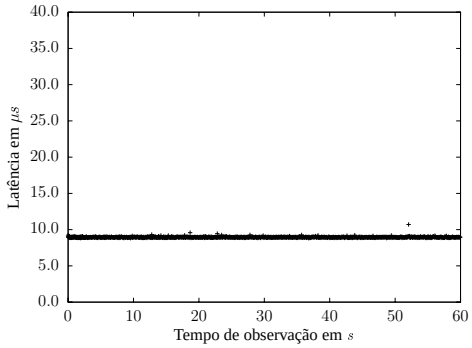
(c) **Linux$^{\mathbf{Prt}}$ - Sem carga**

VM: 21.5, DP: 1.7, Min: 20.3, Max: 45.1



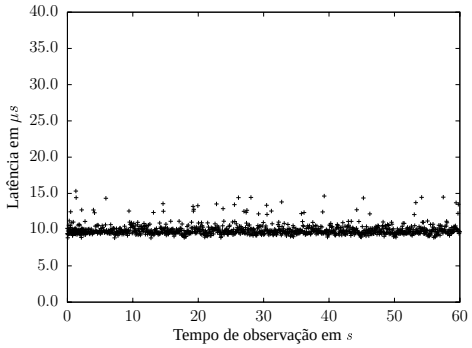
(d) **Linux$^{\mathbf{Prt}}$ - Com carga**

VM: 58.5, DP: 26.4, Min: 17.2, Max: 245.9



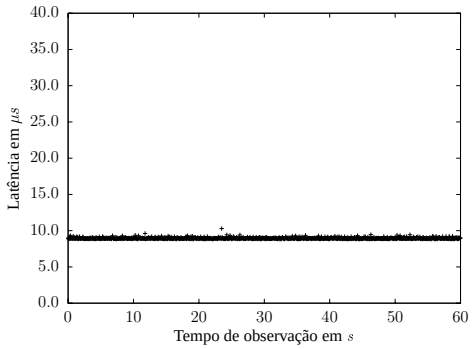
(e) **Linux$^{\mathbf{Rtai}}$ - Sem carga**

VM: 8.7, DP: 0.1, Min: 8.8, Max: 10.7



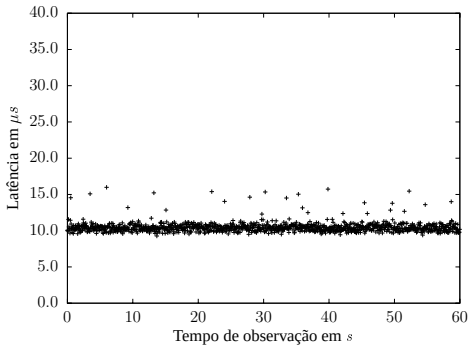
(f) **Linux$^{\mathbf{Rtai}}$ - Com carga**

VM: 10.0, DP: 0.8, Min: 8.8, Max: 20.8



(g) **Linux$^{\mathbf{Xen}}$ - Sem carga**

VM: 9.0, DP: 0.1, Min: 8.8, Max: 11.1



(h) **Linux$^{\mathbf{Xen}}$ - Com carga**

VM: 10.4, DP: 0.7, Min: 8.9, Max: 20.8

Figura 1.5: Latência de interrupção com freqüência de escrita na PP de $20Hz$.

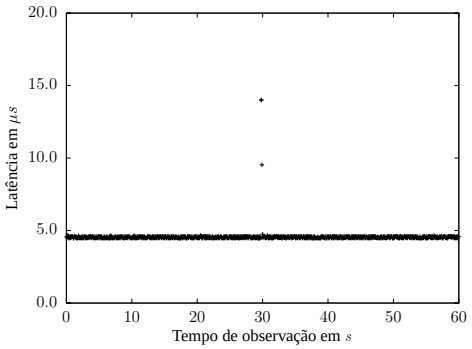
de Linux$^{\mathbf{Std}}$ é inadequado para atender os requisitos de tempo real. Linux$^{\mathbf{Prt}}$, Linux^{Rtai} e Linux$^{\mathbf{Xen}}$, por outro lado, apresentam valores de latências dentro dos padrões esperados. Vale a pena notar o comportamento destes sistemas com carga. Apesar do valores médio encontrados para Linux$^{\mathbf{Xen}}$ $(8,7\mu s)$ser superior ao do Linux$^{\mathbf{Prt}}$ $(3,8\mu s)$ e ao Linux$^{\mathbf{Prt}}$ $(4,4\mu s)$, o desvio padrão é significativamente menor em favor de Linux$^{\mathbf{Xen}}$, característica desejável para sistemas de tempo real críticos.

É interessante ainda observar o comportamento de Linux$^{\mathbf{Prt}}$ sem utilizar o contexto de *threads* de interrupção, isto é, com a opção `IRQF_NODELAY`, comentada anteriormente. Como pode ser observado na figura 4.7, apesar das latências de ativação sem estresse apresentar bons resultados em comparação ao Linux$^{\mathbf{Prt}}$, seus valores com estresse indicam um comportamento menos previsível que o Linux$^{\mathbf{Xen}}$.

1.7 TRABALHOS RELACIONADOS

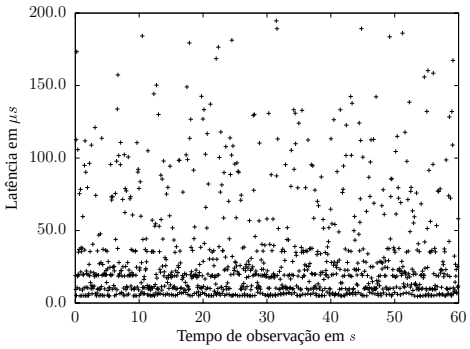
Alguns resultados experimentais comparando Linux$^{\mathbf{Prt}}$ com Linux$^{\mathbf{Std}}$ são apresentados por (ROSTEDT; HART, 2007). Duas métricas são usadas, latências de interrupção e de escalonamento, relacionadas ao escalonamento de uma tarefa periódica. No entanto, os experimentos foram realizados sem sobrecarga do processador e a metodologia usada não foi precisamente descrita. (SIRO; EMDE; MCGUIRE, 2007) realizam um estudo comparativo do Linux$^{\mathbf{Prt}}$, de RT-Linux (V. Yodaiken et al., 2008) e de Linux$^{\mathbf{RTAI}}$ (P. Mantegazza et al., 2008) no qual eles utilizam conjuntamente o *benchmark* LMbench (MCVOY; STAELIN, 1996) e medidas de desvios na execução de uma tarefa periódica. Nestes experimentos, os autores aplicaram uma sobrecarga "média" do processador, sem considerar carga de interrupção. Num outro trabalho, divulgado apenas na Internet (BENOIT; YAGHMOUR, 2005), os desenvolvedores do projeto Adeos apresentam resultados comparativos relativos ao Linux com os *patches* PREEMPT-RT e Adeos. Esta avaliação, bastante abrangente, utiliza o *benchmark* lmbench (MCVOY; STAELIN, 1996) para caracterizar o desempenho das duas plataformas e apresenta resultados de medidas de latências de interrupção realizadas com a porta paralela.

O presente trabalho apresenta resultados de latência de interrupção que confirma os resultados obtidos em (BENOIT; YAGHMOUR, 2005) para a plataforma Linux$^{\mathbf{Xen}}$. Já os resultados encontrados aqui para Linux$^{\mathbf{Prt}}$, sem a opção `IRQF_NODELAY`, diferiram dos apresentados



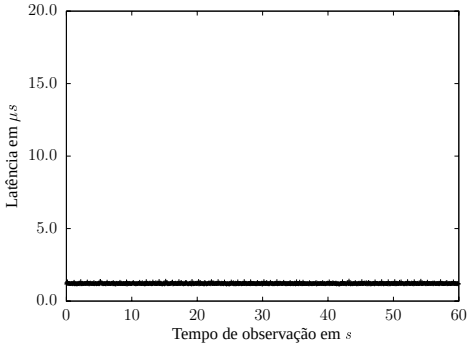
(a) **Linux$^{\mathbf{Std}}$ - Sem carga**

VM: 4.6, DP: 0.4, Min: 4.4, Max: 16.2



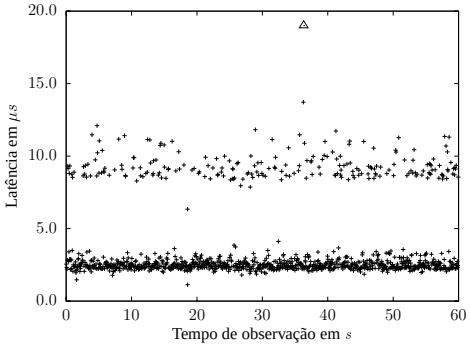
(b) **Linux$^{\mathbf{Std}}$ - Com carga**

VM: 37.3, DP: 48.2, Min: 4.6, Max: 617.5



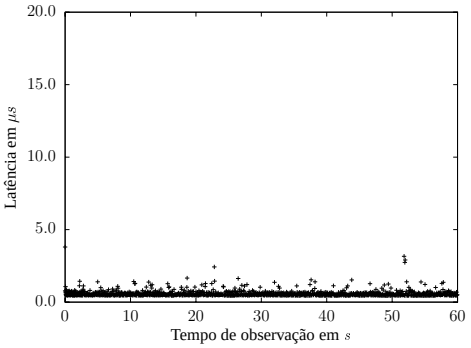
(c) **Linux$^{\mathbf{Prt}}$ - Sem carga**

VM: 2.1, DP: 0.2, Min: 1.2, Max: 9.4



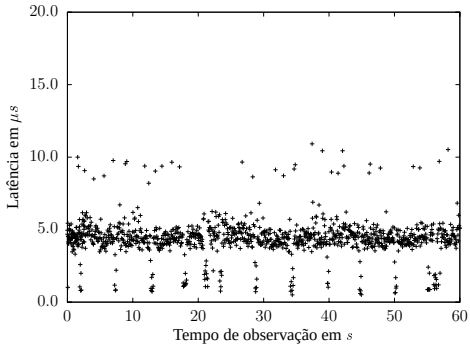
(d) **Linux$^{\mathbf{Prt}}$ - Com carga**

VM: 3.8, DP: 2.8, Min: 1.1, Max: 27.4



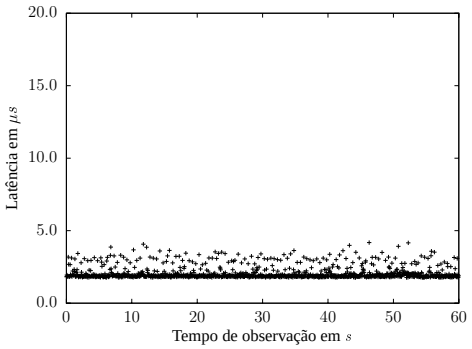
(e) **Linux$^{\mathbf{Rtai}}$ - Sem carga**

VM: 0.6, DP: 0.3, Min: 0.4, Max: 3.8



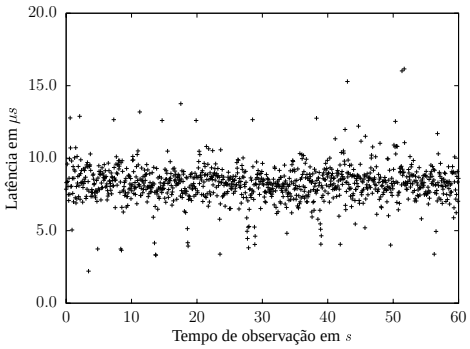
(f) **Linux$^{\mathbf{Rtai}}$ - Com carga**

VM: 4.4, DP: 0.7, Min: 0.5, Max: 14.7



(g) **Linux$^{\mathbf{Xen}}$ - Sem carga**

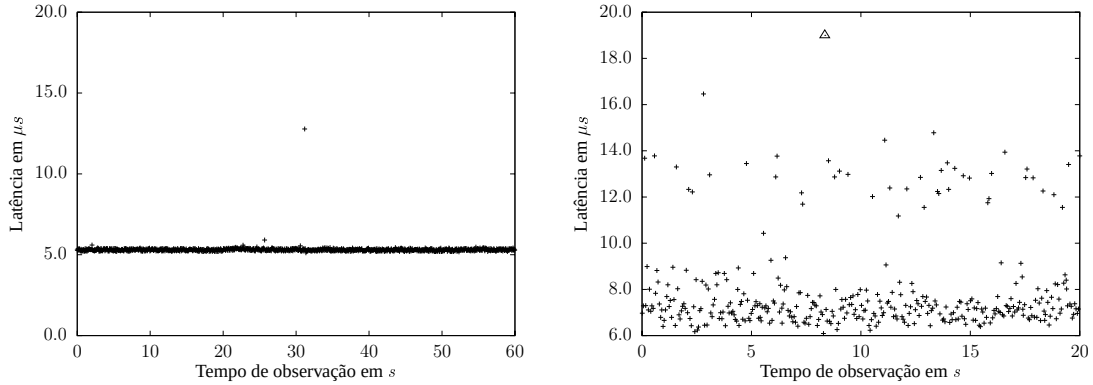
VM: 2.1, DP: 0.5, Min: 1.8, Max: 8.4



(h) **Linux$^{\mathbf{Xen}}$ - Com carga**

VM: 8.7, DP: 0.3, Min: 1.8, Max: 18.7

Figura 1.6: Latência de ativação com freqüência de escrita na PP de $20Hz$.



(a) **Linux$^{\mathbf{Prt}}$ - Sem carga**
VM: 5.3, DP: 0.3, Min: 5.0, Max: 13.1

(b) **Linux$^{\mathbf{Prt}}$ - Com carga**
VM: 8.0, DP: 2.0, Min: 5.2, Max: 31.0

Figura 1.7: Latência de ativação do Linux$^{\mathbf{Prt}}$ desabilitando o *thread* associado as interrupções da PP (opção `IRQF_NODELAY`).

por (BENOIT; YAGHMOUR, 2005), pois uma degradação das garantias temporais por esta plataforma foi observada, tal como visto na seção 4.6.2.1. Em relação às latências de ativação, não temos conhecimento de nenhum outro trabalho comparativo. Experimentos idênticos aos relatados aqui foram conduzidos para a plataforma Linux$^{\mathbf{RTAI}}$ (REGNIER; LIMA; ANDRADE, 2008) e os resultados encontrados são semelhantes aos apresentados para Linux$^{\mathbf{Xen}}$, dado que ambas plataformas utilizam o mesmo *nanokernel*.

1.8 CONCLUSÃO

Neste capítulo, a comparação de três soluções de SOTR baseadas em Linux foi realizada. A metodologia experimental permitiu medir as latências de interrupção e de ativação, em situações de carga variável, tanto do processador quanto de eventos externos tratados por interrupção. Linux padrão apresentou latências no pior caso acima de $100\mu s$, enquanto as plataformas Linux$^{\mathbf{Prt}}$ e Linux$^{\mathbf{Xen}}$ conseguiram prover garantias temporais com uma precisão abaixo de $20\mu s$. No entanto, para se conseguir este comportamento em relação ao Linux$^{\mathbf{Prt}}$, foi necessário desabilitar *threads* de interrupção, tornando o sistema menos flexível. Com tais *threads*, o comportamento de Linux$^{\mathbf{Prt}}$ sofre considerável degradação da sua previsibilidade temporal. A plataforma Xenomai se destacou, pois tanto oferece um ambiente de programação em modo usuário, quanto consegue previsibilidade temporal típica de sistema de tempo real críticos.

REFERÊNCIAS

AUDSLEY, N. C. et al. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, v. 8, n. 5, p. 284–292, 1993.

BACH, M. *The Design of the Unix Operating System*. [S.l.]: Prentice-Hall, 1986.

BARABANOV, M. *A Linux based real-time operating system*. Dissertação (Mestrado) — New Mexico Institution of Mining and Technology, 1997.

BENOIT, K.; YAGHMOUR, K. *Preempt-RT and I-pipe: the numbers*. 2005. Http://marc.info/?l=linux-kernel&m=112086443319815&w=2. Last access 03/08.

BOVET, M. C. D. P. *Understanding the Linux Kernel*. 3rd. ed. [S.l.]: O'Reilly, 2005.

CALANDRINO, J. et al. Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *In Proceedings of the 27th IEEE Real-Time Systems Symposium*. [S.l.: s.n.], 2006. p. 111–126.

DIJKSTRA, E. Solution of a problem in concurrent programming and control. *Communications of the ACM*, v. 8, n. 9, p. 569, September 1965.

DOZIO, L.; MANTEGAZZA, P. Linux real time application interface (RTAI) in low cost high performance motion control. In: *Proceedings of the conference of ANIPLA, Associazione Nazionale Italiana per l'Automazione*. [S.l.: s.n.], 2003.

FRY, G.; WEST, R. On the integration of real-time asynchronous event handling mechanisms with existing operating system services. In: *Proceedings of the International Conference on Embedded Systems and Applications (ESA'07)*. [S.l.: s.n.], 2007.

GERUM, P. *Life with ADEOS*. 2005. www.xenomai.org/documentation/branches/v2.0.x/pdf/Life-with-Adeos.pdf. Last access 01 / 2008.

GORZ, A. *Écologie et Liberté*. [S.l.]: Galilée, 1977.

I. Molnar et al. *PreemptRT*. 2008. http://rt.wiki.kernel.org - Last access jan. 08.

IEEE. *IEEE Standart 1003.1 (POSIX), 2004 Edition*. 2004.

L. Torvalds et al. *Kernel*. 2008. http://www.kernel.org - Last access jan. 08.

LAMPORT, L. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, v. 17, n. 8, p. 453–455, AUGUST 1974.

LAMPORT, L.; MELLIAR-SMITH, M. Real-time model checking is really simple. In: BORRIONE, I. D.; PAUL, W. J. (Ed.). *Correct Hardware Design and Verification Methods*. [S.l.]: Springer-Verlag, 2005. (LNCS, v. 3725), p. 162–175.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogram in a hard real-time environment". *Journal of ACM*, v. 20, n. 1, p. 40–61, 1973.

MCKENNEY, P. *A realtime preemption overview*. 2005. Http://lwn.net/Articles/146861/ - Last access dez. 07.

MCVOY, L. W.; STAELIN, C. lmbench: Portable tools for performance analysis. In: *USENIX Annual Technical Conference*. [S.l.: s.n.], 1996. p. 279–294.

MELLOR-CRUMMEY, J. M.; SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 9, n. 1, p. 21–65, 1991.

MOLNAR, I. *kernel/timer.c design*. 2005. Internet, http://lkml.org/lkml/2005/10/19/46 - Last access 01/08.

OLIVEIRA, R. S. de; CARISSIMI, A. da S.; TOSCANI, S. S. *Sistemas Operacionais*. [S.l.]: SagraLuzzatto, 2001. ISBN 85-241-0643-3.

P. Gerum et al. *Xenomai*. 2008. http://www.xenomai.org - Last access jan. 08.

P. Mantegazza et al. *RTAI*. 2008. http://www.rtai.org - Last access jan. 08.

PICCIONI, C. A.; TATIBANA, C. Y.; OLIVEIRA, R. S. de. *Trabalhando com o Tempo Real em Aplicações Sobre o Linux*. Florianópolis -SC, Dezembro 2001.

QI, X.; PARMER, G.; WEST, R. An efficient end-host architecture for cluster communication services. In: *Proceedings of the IEEE International Conference on Cluster Computing (Cluster '04)*. [S.l.: s.n.], 2004.

RAYNAL, M. *Algorithms for Mutual Exclusion*. Massachusetts, Cambridge: MIT Press, 1986.

REGNIER, P.; LIMA, G.; ANDRADE, A. TLA+ formal specification and verification of a real-time ethernet protocol. Submitted to SBMF. 2008.

ROSTEDT, S.; HART, D. V. Internals of the rt patch. In: *Proceedings of the Linux Symposium*. [S.l.: s.n.], 2007. p. 161–172.

SANTOS, M. *Por uma outra globalização - do pensamento único à consciência universal*. São Paulo: Record, 2000.

SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. P. Priority Inheritance Protocols: An approach to real-time synchronisation. *IEEE Transaction on Computers*, v. 39, n. 9, p. 1175–1185, 1990.

SIRO, A.; EMDE, C.; MCGUIRE, N. Assessment of the realtime preemption patches (RT-Preempt) and their impact on the general purpose performance of the system. In: *Proceedings of the 9th Real-Time Linux Workshop*. [S.l.: s.n.], 2007.

SRINIVASAN, B. et al. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In: *Proc. of the Real-Time Technology and Applications Symposium*. [S.l.: s.n.], 1998.

STODOLSKY, D.; CHEN, J.; BERSHAD, B. Fast interrupt priority management in operating systems. In: *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*. [S.l.: s.n.], 1993. p. 105–110.

TANENBAUM, A. *Modern Operating Systems*. [S.l.]: Prentice-Hall, 2001. ISBN 85-241-0643-3.

TINDELL, K.; BURNS, A.; WELLINGS, A. J. An extendible approach for analysing fixed priority hard real-time tasks". *Real-Time Systems*, v. 4, n. 2, p. 145–165, 1994.

V. Yodaiken et al. *RT-Linux*. 2008. http://www.rtlinuxfree.com - Last access jan. 08.

YAGHMOUR, K. The real-time application interface. In: *Proceedings of the Linux Symposium*. [S.l.: s.n.], 2001.