Adaptações do Linux para Suportar Aplicações com Requisitos de Tempo Real

Rômulo Silva de Oliveira

LCMI-DAS - Universidade Federal de Santa Catarina
Caixa Postal 476, Florianópolis-SC, 88040-900

romulo@das.ufsc.br

Abstract. *Real-time applications are built more easily if they can take advantage of the services of an operating system. This paper initially discusses constructive aspects of operating systems and their implications in the perspective of real-time theory. After, five Linux adaptations to support real-time applications are described.*

Resumo. *Aplicações de tempo real são mais facilmente construídas se puderem aproveitar os serviços de um sistema operacional. Este artigo inicialmente discute aspectos construtivos de sistemas operacionais e suas implicações sob a ótica da teoria de tempo real. Em seguida, são descritas cinco adaptações do Linux para suportar aplicações de tempo real.*

1. Introdução

Na medida que o uso de sistemas computacionais prolifera em nossa sociedade, aplicações com requisitos de tempo real tornam-se cada vez mais comuns. Estas aplicações variam muito com relação ao tamanho, complexidade e criticalidade. Entre os sistemas mais simples estão os controladores embutidos em utilidades domésticas, tais como lavadoras de roupa e videocassetes. Na outra extremidade deste espectro estão os sistemas militares de defesa e o controle de tráfego aéreo. Exemplos de aplicações críticas são os sistemas responsáveis pela monitorização de pacientes em hospitais e os sistemas embarcados em veículos, de automóveis até aviões e sondas espaciais. Entre aplicações não críticas estão os videogames e as aplicações multimedia.

Entre os sistemas de tempo real podemos destacar aqueles identificados como sistemas tempo real embutidos (*embedded real-time systems*). Um sistema computacional embutido corresponde a um ou mais microprocessadores, um sistema operacional e um software aplicativo que ficam inseridos em um produto maior para processar as funções de controle deste produto. O projeto de um sistema computacional de propósito geral deve considerar as possíveis necessidades de um enorme espectro de usuários. Diferentemente, um sistema computacional embutido deve suportar apenas um conjunto restrito de funções, definidas pelo equipamento maior no qual ele está inserido.

Uma vez que tanto a aplicação como o sistema operacional (SO) compartilham os mesmos recursos do hardware, o comportamento temporal do SO afeta o comportamento temporal da aplicação. Por exemplo, considere a rotina do sistema operacional que trata as interrupções do relógio de hardware. O projetista da aplicação pode ignorar completamente a função desta rotina, mas não pode ignorar o seu efeito temporal, isto é, a interferência que ela causa na aplicação.

A teoria de escalonamento tempo real teve um grande avanço nos últimos anos. Apesar disto, ainda não é corrente o uso da teoria de tempo real na modelagem e análise de sistemas operacionais complexos. Mais do que isto, aplicações com deadlines críticos ainda hoje empregam apenas pequenos núcleos de tempo real, que implementam um conjunto limitado de serviços, mas são capazes de oferecer previsibilidade determinista. Sistemas operacionais com funcionalidade completa, incluindo sistema de arquivos, interface gráfica de usuário e protocolos de comunicação, suportam apenas tempo real brando.

Este artigo investiga as práticas normalmente usadas na construção de sistemas operacionais. Busca-se determinar os efeitos dos mecanismos empregados, ao mesmo tempo que são feitas indicações sobre quais mecanismos são mais apropriados para a construção de um sistema operacional que deverá suportar aplicações com requisitos de tempo real. São também descritas diversas variações do sistema operacional Linux para melhor atender as demandas de tempo real.

Algumas questões de modelagem discutidas aqui apareceram antes na literatura [AUDSLEY 1993]. Pesquisas recentes focaram alguns dos aspectos construtivos abordados neste artigo. Em [ABENI 2002] são discutidos temporizadores e preempção. Em [CARLSSON 2002] é considerado o problema da determinação do tempo de execução no pior caso do código encontrado em sistemas operacionais. Em [MEHNERT 2002] é analisado o custo de manter espaços de endereçamento separados.

O propósito deste trabalho é colaborar para a análise qualitativa dos núcleos dos sistemas operacionais que pretendem suportar aplicações de tempo real. Em especial, aqueles sistemas operacionais derivados do Linux. Em [OLIVEIRA 2003] aparece uma versão preliminar dos aspectos construtivos, sem os estudos de caso. A seção 2 discute vários aspectos construtivos de SO sob a luz da teoria de escalonamento. A seção 3 descreve diversas adaptações feitas no sistema operacional Linux no sentido do mesmo suportar aplicações de tempo real. A seção 4 apresenta as conclusões.

2. Análise dos Aspectos Construtivos dos Sistemas Operacionais

Esta seção analisa de que maneiras o sistema operacional impacta o tempo de resposta das tarefas de tempo real. Em outras palavras, quais os aspectos mais relevantes na construção de um sistema operacional de tempo real.

2.1 Algoritmo de Escalonamento Apropriado

Ao considerar como o sistema operacional pode afetar o comportamento da aplicação no tempo, o primeiro aspecto lembrado é o algoritmo de escalonamento. Do ponto de vista da análise de escalonabilidade, deseja-se um algoritmo para o qual a literatura ofereça um método de análise e testes de escalonabilidade. A literatura é bastante rica nesse sentido, oferecendo um grande leque de possibilidades. No contexto da teoria baseada em prioridades preemptivas, isto não é um problema, pois este é o algoritmo implementado pela maioria dos sistemas operacionais de tempo real.

2.2 Níveis de Prioridade Suficientes

Embora o escalonamento baseado em prioridades seja suportado pela maioria dos sistemas operacionais, o número de diferentes níveis de prioridade varia bastante. Por exemplo, o padrão POSIX da IEEE exige no mínimo 32 níveis de prioridade. Em

[CAYSSIALS 1999] é mostrado que, quando o número de níveis de prioridade disponíveis é menor do que o número de tarefas, passa a ser necessário agrupar várias tarefas no mesmo nível, o que diminui a escalonabilidade do sistema. Ainda em [CAYSSIALS 1999] é apresentado um algoritmo para calcular o número mínimo de níveis de prioridade onde o sistema ainda é escalonável, quando Taxa Monotônica é usada como política de atribuição de prioridades. Em [AUDSLEY 2001] o mesmo problema é atacado, mas a partir de um algoritmo ótimo para atribuição de prioridades. Em termos de modelagem, o ideal é que o número de níveis de prioridade seja igual ao número de tarefas no sistema.

2.3 Alteração das Prioridades pelo Sistema Operacional

Muitos sistemas operacionais manipulam por conta própria as prioridades das tarefas. Por exemplo, o mecanismo de envelhecimento (*aging*) é usado por vezes para aumentar temporariamente a prioridade de uma tarefa que, por ter prioridade muito baixa, nunca consegue executar. Muitos sistemas operacionais de propósito geral (SOPG) também incluem mecanismos que reduzem automaticamente a prioridade de uma *thread* na medida que ela consome tempo de processador. Este mecanismo é utilizado para favorecer as tarefas com ciclos de execução menor e diminuir o tempo médio de resposta no sistema.

Esses mecanismos fazem sentido em um SO de propósito geral, quando o objetivo é estabelecer um certo grau de justiça entre tarefas e evitar postergação indefinida. No contexto de tempo real não existe preocupação com justiça na distribuição dos recursos mas sim com o atendimento dos deadlines. Para efeitos de análise, esses mecanismos, além de não contribuir para a qualidade temporal, tornam mais complexo o esforço de modelagem e devem ser evitados.

2.4 Tratadores de Interrupções

Parte importante de qualquer sistema operacional é a execução dos tratadores de interrupção. Eles são responsáveis pela atenção do sistema a eventos externos, como alarmes ou a passagem do tempo. Ao mesmo tempo, eles são uma importante fonte de interferência sobre as demais tarefas. Para efeito de modelagem, cada tratador de interrupção é considerado como tarefa de prioridade mais alta do que tarefas normais.

Muitos tratadores de interrupção apresentam comportamento esporádico, como aqueles associados com teclados, controladores de disco e de rede. Embora difícil, o estabelecimento de um intervalo mínimo entre ativações para esses tratadores é uma necessidade matemática para a análise quantitativa do sistema. A prática corrente dos SOPG é não colocar restrições quanto a freqüência de interrupções de qualquer tipo. Entretanto, a pseudo-tarefa tratador de interrupção tem uma alta prioridade no sistema e sua capacidade de gerar interferência é grande. Ela deve ser controlada em sistemas operacionais para aplicações de tempo real.

2.5 Latência dos Tratadores de Interrupções

O tempo entre a sinalização de uma interrupção no hardware e o inicio da execução de seu tratador é normalmente chamado de latência do tratador de interrupção. A latência no disparo de um tratador de interrupção inclui o tempo que o hardware leva para realizar o processamento de uma interrupção, isto é, salvar o contexto atual (*program*

counter e *flags*) e desviar a execução para o código do tratador. Também é necessário incluir o tempo máximo que as interrupções podem ficar desabilitadas. Por vezes, trechos de código do sistema operacional precisam executar com as interrupções desabilitadas. Por exemplo, quando é acessada uma estrutura de dados também usada por tratadores de interrupção.

O atraso no reconhecimento da interrupção pode ser modelado como um *release jitter* associado com cada pseudo-tarefa "tratador de interrupção". Muitas arquiteturas associam prioridades aos diversos tipos de interrupção. Dessa forma, uma interrupção de alto nível pode suspender temporariamente a execução do tratador da interrupção de baixo nível. Esta situação é coerente com a idéia de uma pseudo-tarefa de prioridade alta interferindo com uma pseudo-tarefa de prioridade baixa.

Por outro lado, o início do tratador de interrupção de baixo nível pode ser obrigado a esperar a conclusão do tratador da interrupção de alto nível. Para efeitos de modelagem, uma pseudo-tarefa associada com um dado tratador de interrupção recebe interferência quando é atrapalhada por um tratador de interrupção mais prioritária, mas sofre *release jitter* quando atrasa em função de uma tarefa de prioridade mais baixa desabilitar interrupções.

2.6 *Threads* de Kernel

Muitos sistemas operacionais incluem *threads* de kernel com execução periódica, responsáveis pelas tarefas de manutenção. Por exemplo, escrever partes da *cache* do sistema de arquivos para o disco, atualizar contabilizações, etc. É importante que essas *threads* de kernel façam parte do esquema global de prioridades. Dado o caráter essencial de algumas dessas atividades, pode ser necessário que elas possuam prioridade superior às tarefas da aplicação. Isto não é um problema, desde que fique bem caracterizado quais são essas tarefas, com o período, o tempo máximo de computação e a prioridade de cada uma.

Uma solução paliativa, adotada em sistemas que não suportam *threads* de kernel, é "anotar trabalhos a serem feitos" em listas, e depois executá-los em momentos pré-estabelecidos, tais como uma chamada de sistema ou uma interrupção de hardware. Esta solução de projeto dificulta a modelagem, pois o "trabalho" em questão representa uma carga que executa não conforme a sua prioridade, mas conforme a dinâmica do sistema. Por exemplo, se um "trabalho" for executado no momento que a tarefa executando faz uma chamada de sistema, ele terá a prioridade da tarefa executando. Se ele for executado na próxima interrupção de relógio, ele terá a prioridade da pseudo-tarefa tratador do relógio. Esse esquema efetivamente cria tarefas não só com prioridades variáveis, mas com prioridades que variam conforme a dinâmica do sistema, tornando muito difícil qualquer análise.

2.7 Implementação das Filas

Tradicionalmente, são utilizados algoritmos e estruturas de dados convencionais, tais como listas encadeadas, na implementação das filas dentro dos sistemas operacionais, por exemplo filas de espera e a própria fila do processador. O processamento de listas encadeadas introduz substancial *overhead* em cenários de pior caso, como na liberação simultânea de várias tarefas periódicas. A complexidade computacional desta operação está associada com $O(n^2)$, pois no pior caso é necessário mover n tarefas da lista

encadeada de espera (por exemplo, a lista de espera pela passagem de tempo) para a lista encadeada que representa a fila do processador [KATCHER 1993].

Segundo [ANGELOV 2002], o problema do *overhead* associado com a liberação de tarefas pode ser minimizado através da substituição da tradicional implementação da fila do processador como uma lista encadeada. No lugar, a fila do processador pode ser pensada como um conjunto de vetores booleanos, resultando em operações rápidas e tempo de execução constante, independente do número de tarefas envolvidas.

2.8 Tempo de Chaveamento entre Tarefas

Uma métrica muito citada no mercado de sistemas operacionais é o tempo para chaveamento de contexto entre duas tarefas. Este tempo inclui salvar os registradores da tarefa que está executando e carregar os registradores com os valores da nova tarefa, incluindo qualquer informação necessária para a MMU (*memory management unit*) funcionar corretamente. Em geral, esta métrica não inclui o tempo necessário para decidir qual tarefa vai executar, uma vez que isto depende do algoritmo de escalonamento utilizado.

Em termos de modelagem, o tempo de chaveamento pode ser somado ao tempo máximo de execução de cada tarefa. Necessariamente, cada ativação de cada tarefa deverá carregar o seu contexto. Se a tarefa em questão é preemptada por outra, a interferência que ela sofre da outra incluirá o tempo de chaveamento de contexto, o qual foi também somado no tempo máximo de execução da outra tarefa.

2.9 Temporizadores de Alta Resolução

Sistemas operacionais em geral oferecem temporizadores para as tarefas da aplicação, que armam temporizações ao final das quais uma determinada ação ocorre. Essa ação pode ser o envio de um sinal Unix ou a liberação da tarefa após um *sleep*. Temporizadores são utilizados na implementação de *time-out* e tarefas periódicas.

Nos SOPG temporizadores são implementados através de interrupções periódicas geradas por um relógio de hardware. Por exemplo, a cada 10ms uma interrupção é gerada. Essa implementação pode gerar erros grosseiros. Suponha que um sleep(15ms) é solicitado imediatamente após a ocorrência de uma interrupção do relógio. A tarefa em questão vai esperar 10ms até a próxima interrupção, quando o sistema inicia a contagem do tempo. E deverá esperar mais 20ms, pois as interrupções acontecem de 10 em 10ms. Ao final, a espera de 15ms consumiu um total de 30ms.

SOTR utilizam temporizadores de alta resolução, baseados em interrupções aperiódicas. O relógio de hardware não gera interrupções periódicas mas é programado para gerar uma interrupção no próximo momento de interesse. Considerando o exemplo anterior e supondo ser esta espera o próximo evento de interesse, o relógio do hardware seria programado para gerar uma interrupção exatamente no momento esperado pela tarefa em questão.

O tratador de interrupções de relógio periódicas pode ser modelado como uma tarefa periódica cujo período é o intervalo entre interrupções, o tempo de execução é a duração do código do tratador e o *release jitter* corresponde a latência de interrupções do sistema. Entretanto, o temporizador de alta resolução não é periódico, mas sim

esporádico. Além disso, seu intervalo mínimo entre ativações depende da dinâmica das tarefas do sistema. Como esse tratador é sempre executado em função de uma temporização solicitada por uma tarefa, é mais apropriado modelá-lo associado com as próprias tarefas. Para efeitos de análise temos uma relação de precedência entre a pseudo-tarefa tratador da interrupção e a tarefa executada depois. O período não é determinado pelo tratador mas sim pela tarefa que solicita a temporização. A latência de interrupções continua sendo um *release jitter* para o tratador. O tempo de computação necessário para manipular as filas é associado com o tratador, enquanto o tempo necessário para carregar o contexto da tarefa é somado ao tempo de execução dela. As prioridades são diferentes, pois o tratador possui prioridade alta no sistema, enquanto a tarefa em si pode possuir qualquer prioridade, coerente com o fato de a execução da tarefa liberada poder não ser imediata, dependendo da prioridade da tarefa em execução.

2.10 Comportamento das Chamadas de Sistema no Pior Caso

Para as aplicações de tempo real, o tempo de execução no pior caso é mais relevante do que o tempo de execução no caso médio. Em geral, a implementação das chamadas de sistema é feita de maneira a minimizar o tempo médio. Aplicações de tempo real são beneficiadas quando o código que implementa as chamadas de sistema apresenta bom comportamento também no pior caso. Na construção de um SOTR devem ser evitados algoritmos que apresentam excelente comportamento médio porém um péssimo comportamento de pior caso decorrente de uma situação cuja probabilidade de ocorrer é muito baixa, porém não nula.

Busca-se aqui reduzir o tempo máximo de execução de uma tarefa que, como parte de sua execução, faz uma chamada de sistema. O maior problema está na dependência que pode haver entre o tempo de execução da chamada de sistema e o estado do SO quando a chamada é feita. Quanto mais complexo for o SO, mas difícil é calcular este tempo.

2.11 Preempção de Tarefa Executando Código do Sistema

Um kernel não preemptivo é capaz de gerar grandes inversões de prioridade. Suponha que uma tarefa de baixa prioridade faça uma chamada de sistema e, enquanto o código do kernel é executado, ocorre a interrupção de hardware que deveria liberar uma tarefa de alta prioridade. Nesse tipo de kernel a tarefa de alta prioridade terá que esperar até que a chamada de sistema da tarefa de baixa prioridade termine, para então executar. Temos então que o kernel está executando antes o pedido de baixa prioridade, em detrimento da tarefa de alta prioridade.

Um kernel com pontos de preempção reduz o problema, mas ainda permite a inversão de prioridades, quando a chamada de sistema de baixa prioridade prossegue sua execução em detrimento da tarefa de alta prioridade até encontrar um ponto de preempção. Um SOTR deve ser preemptivo, ainda que em determinados momentos interrupções precisem ser desabilitadas em função das estruturas de dados acessadas.

2.12 Mecanismos de Sincronização Apropriados

A literatura de tempo real é rica em mecanimos de sincronização apropriados para tempo real, como mutexes que incorporam herança de prioridade, *priority ceiling*, etc. Recursos semelhantes existem para sincronização tempo real baseada em mensagens,

tais como mensagens com prioridades, *priority ceiling* para *threads* tratando mensagens, etc. Tais mecanismos reduzem a inversão de prioridades dentro do kernel, reduzindo o tempo de bloqueio sofrido por tarefas que fazem chamadas de sistema.

2.13 Granularidade das Seções Críticas dentro do Kernel

Um kernel preemptivo é capaz de chavear imediatamente para a tarefa de alta prioridade quando o mesmo é liberado. Entretanto, inversão de prioridade dentro do kernel ainda é possível quando a tarefa de alta prioridade recém ativada faz uma chamada de sistema mas é bloqueada em função da necessidade de acessar, dentro do kernel, uma estrutura de dados compartilhada que foi anteriormente alocada por uma tarefa de mais baixa prioridade. Mesmo mecanismos apropriados de sincronização não conseguem evitar esta situação, decorrente do fato de duas tarefas acessarem a mesma estrutura de dados.

Manter uma granularidade fina para as seções críticas dentro do kernel, embora aumente a complexidade do código, reduz o tempo que uma tarefa de alta prioridade precisa esperar até que uma tarefa de baixa prioridade libere a estrutura de dados compartilhada. Em termos de modelagem, temos a redução do tempo de bloqueio sofrido pela tarefa mais prioritária.

2.14 Gerência de Recursos em Geral

Todos os sistemas operacionais desenvolvidos ou adaptados para tempo real mostram grande preocupação com a divisão do tempo do processador entre as tarefas. Entretanto, o processador é apenas um recurso do sistema. Memória, periféricos, controladores, servidores também deveriam ser escalonados visando atender os requisitos temporais da aplicação. Muitos sistemas ignoram isto e tratam os demais recursos da mesma maneira empregada por um SOPG, isto é, tarefas são atendidas pela ordem de chegada.

Todas as filas do sistema deveriam respeitar as prioridades das tarefas, e não apenas a fila do processador. Por exemplo, as requisições de disco deveriam ser ordenadas conforme a prioridade e não pela ordem de chegada ou para minimizar o tempo de acesso. Quando um recurso é acessado pela ordem de chegada temos a possibilidade de bloqueio da tarefa mais prioritária pela menos prioritária. Este bloqueio pode ser devastador para a escalonabilidade, pois sua duração está associada com toda a fila de tarefas que existia quando a tarefa de alta prioridade solicitou o recurso.

2.15 Tratadores de Dispositivos (*Device-Drivers*)

Por vezes a execução de uma tarefa de alta prioridade é suspensa temporariamente, quando ocorre uma interrupção de periférico. O processador passa a executar o tratador de interrupção incluído em *device-driver* associado com o periférico em questão. Muitos sistemas não limitam o tempo de execução desse tratador, permitindo na prática que o código associado com o *device-driver* em questão tenha uma prioridade maior do que qualquer tarefa no sistema e execute por quanto tempo quiser. Em termos de modelagem, isto significa uma tarefa com tempo de execução possivelmente longo e alta prioridade, gerando interferência em todas as tarefas do sistema, sendo sua alta prioridade decorrente não de uma política de prioridades explícita, mas de um efeito colateral da construção do SO.

Em um SOTR, os tratadores de interrupção associados com *device-drivers* simplesmente devem liberar *threads* de kernel as quais seriam responsáveis pela

execução do código que efetivamente responde ao sinal do periférico. Fazendo com que as *threads* de kernel respeitem a estrutura de prioridades do sistema, fica restaurado o desejo do programador com respeito aos tempos de resposta das tarefas.

Por exemplo, o teclado pode ser considerado um periférico de baixa prioridade. Caso aconteça uma interrupção de teclado durante a execução da tarefa de alta prioridade, o tratador de interrupções do teclado simplesmente coloca a *thread* "Atende Teclado" na fila do processador e retorna. A tarefa de alta prioridade pode concluir sua execução e depois disso o atendimento ao teclado propriamente dito será feito pela *thread* correspondente.

3. Exemplos de Suportes para Tempo Real Baseados no Linux

Existem dezenas senão centenas de suportes para tempo real, nos mais variados níveis de preço, robustez, funcionalidade e previsibilidade temporal. O conjunto de soluções escolhido para ser apresentado aqui procura ilustrar os tópicos apresentados nas seções anteriores, sem ter a pretenção de ser um levantamento preciso do mercado. Serão descritas variações do Linux para tempo real, cada uma alterando o kernel convencional do Linux de uma maneira específica.

Linux é um sistema operacional com fonte aberto, estilo Unix, originalmente criado por Linus Torvalds a partir de 1991 com o auxílio de desenvolvedores espalhados ao redor do mundo [BOVET 2001]. Linux é *"free software"* no sentido que pode ser copiado, modificado, usado de qualquer forma e redistribuído sem nenhuma restrição. Entretanto, ninguém criando uma adaptação do Linux pode tornar o produto resultante proprietário, pois o mesmo é desenvolvido sob o *"GNU General Public License"*.

O Linux convencional segue o estilo de um kernel Unix tradicional, não baseado em microkernel, e não apropriado para aplicações de tempo real [EULER 1999]. Entretanto, o kernel do Linux possui um recurso que facilita sua adaptação para o contexto de tempo real. Embora o kernel ocupe um único espaço de endereçamento, ele aceita módulos carregáveis em tempo de execução, os quais podem ser incluídos e excluídos do kernel sob demanda. Estes módulos executam em modo privilegiado e são usados normalmente na implementação de tratadores de dispositivos (*device-drivers*), sistemas de arquivos e protocolos de rede. No caso dos sistemas de tempo real, esta característica facilita a transferência de tecnologia da pesquisa para a prática. Soluções de escalonamento tempo real podem ser implantadas dentro do kernel de um sistema operacional de verdade. Como o código fonte do Linux é aberto, é possível estudar o seu comportamento temporal, algo que é impossível com SOTR comerciais cujo kernel é tipicamente uma caixa preta.

3.1 KURT Linux

O KURT-Linux [SRINIVASAN 1998], ou "Kansas University Real Time Linux" é um sistema operacional de tempo real que permite o escalonamento explícito e relativamente preciso no tempo de qualquer evento, inclusive a execução de tarefas com restrições de tempo real.

KURT-Linux é uma modificação do kernel convencional, onde destaca-se o aumento na precisão da marcação da passagem do tempo real através do emprego de temporizadores com alta resolução. Como conseqüência, tem-se uma maior precisão nas

ações associadas com a passagem do tempo. Uma importante constatação foi que a programação explícita do relógio de hardware para cada evento de interesse, usando uma resolução de microsegundos, gerou uma carga adicional pequena para o sistema. Várias técnicas foram usadas em conjunto para obter maior precisão de relógio e manter funcionando o código do kernel que supõe interrupções periódicas de relógio.

No KURT-Linux, módulos de tempo real pertencentes a aplicação são incorporados ao kernel. Eles executam em modo kernel e podem acessar todos os recursos do sistema. O escalonamento é feito através de uma lista que informa qual rotina presente em um módulo de tempo real deve executar quando. Toda rotina de tempo real executa a partir da hora marcada e suspende a si própria usando uma chamada de sistema apropriada. Desta forma, a precisão do relógio é transferida para o comportamento das tarefas. O sistema operacional supõe que os módulos de tempo real são bem comportados e não vão exceder o tempo de processador previamente alocado. Embora este tipo de escalonamento seja menos flexível do que o baseado em prioridades, ele ainda é suficiente para um grande conjunto de aplicações. Os autores do KURT-Linux atestam que nunca foi o objetivo do projeto suportar todos os algoritmos de escalonamento tempo real presentes na literatura.

Tarefas de tempo real periódicas também são aceitas. Elas definem o período desejado e são ativadas pelo KURT-Linux no início de cada período. Quando a tarefa completa uma dada ativação, ela suspende a si mesma, para ser acordada pelo sistema no início do próximo período. Uma extensão ao modelo original permite que uma tarefa programada como um laço infinito (um servidor) execute em determinados momentos previamente reservados. Por exemplo, execute 100 microsegundos dentro de cada milisegundo. Soluções de escalonamento baseadas no conceito de utilização do processador podem ser implementadas através deste mecanismo.

KURT-Linux é capaz de prover uma excelente precisão com respeito ao instante de liberação das tarefas de tempo real. Através da construção e manutenção de uma escala de execução apropriada, a interferência entre tarefas de tempo real pode ser resolvida. Entretanto, o esquema adotado é limitado pelo código convencional do kernel do Linux, o qual é capaz de gerar bloqueios e inversões de prioridade significativas, quando uma tarefa de tempo real solicita algum serviço. Também os *devices-drivers* são capazes de reduzir a precisão do sistema através da desabilitação de interrupções. Seu momento de execução não é definido pela escala construída, mas sim pelas interrupções de hardware. Mesmo os processos convencionais do Linux podem atrapalhar a execução das tarefas de tempo real, quando utilizam recursos do kernel que essas necessitam. O sistema permite que a execução de processos Linux convencionais seja proibida, reduzindo assim as pertubações sobre os tempo de resposta das tarefas de tempo real.

KURT-Linux está baseado no conceito de escala de execução. Mas, constrói a escala de execução sem considerar as ativações assíncronas dos tratadores de interrupções associados com *devices-drivers*. Também ignora os bloqueios possíveis em função de recursos compartilhados dentro do kernel. Logo, o comportamento do sistema será tanto melhor quanto as tarefas não utilizarem os serviços do kernel do Linux e periféricos não forem utilizados. Seu comportamento ideal acontece quando as tarefas de tempo real realizam suas computações sem fazer chamadas de sistema e *device-drivers* não incluem tratadores de interrupções. Em [SRINIVASAN 1998] é sugerido

que os construtores de aplicações selecionem para uso apenas os subsistemas do Linux cujo efeito sobre o comportamento das tarefas de tempo real seja tolerável.

3.2 Linux/RK

Linux/RK foi desenvolvido no Real-time and Multimedia Systems Laboratory na Carnegie Mellon University. Linux/RK implementa no Linux o conceito de *resource kernel* [OIKAWA 1998] [OIKAWA 1999]. Ele insere no kernel convencional do Linux um módulo chamado *Portable Resource Kernel*, o qual controla o acesso aos recursos físicos, provendo garantias e ao mesmo tempo impondo limites a esses acessos. Basicamente, aplicações devem solicitar reservas para determinadas quantidades de cada recurso e o kernel então garante que os recursos associados com as reservas confirmadas estarão disponíveis no momento certo. Em termos de implementação, Linux/RK corresponde a um módulo carregado no kernel, além da inserção de chamadas em pontos específicos do kernel Linux convencional. O conceito de resource kernel não é novo. Existem trabalhos que implementam esta idéia no RT-Mach [RAJKUMAR 1998], para processador, disco e rede local.

No contexto de um *resource kernel*, uma reserva representa uma fatia de algum recurso computacional, tal como tempo de processador, memória física ou tempo do controlador de disco. A entidade "reserva" é implementada pelo kernel, o qual monitora o seu uso, garantindo a disponibilidade dos recursos para atender as reservas feitas, ao mesmo tempo que impede o uso dos recursos além dos limites das reservas. Alguns recursos podem ser multiplexados no tempo, como processador, meio de comunicação e acesso ao disco. Cada reserva de recurso multiplexado no tempo é caracterizada por um período P, um tempo de utilização C e um deadline D, sendo $D \leq P$. Recursos como espaço em memória são dedicados, isto é, não são multiplexados no tempo.

Um conjunto de reservas (*resource set* é usado no original) é associado com um ou mais programas e permite o uso exclusivo da quantidade reservada de recursos por estes programas. Um conjunto de reservas agrupa as reservas para os programas associados, permitindo que o kernel detecte qualquer uso indevido de recursos por parte da aplicação. O conjunto de reservas é um conceito mais apropriado para o nível da aplicação, além de facilitar a implementação dos mecanismos. Por exemplo, cada processo Linux está associado com um conjunto de reservas. O código de aplicação acessa as facilidades do módulo RK através de uma interface composta pelas funções: cria conjunto de reservas, destrói conjunto de reservas, associa processo com conjunto de reservas, desassocia processo de um conjunto de reservas, cria uma reserva.

No contexto do Linux/RK, o módulo RK inserido no kernel convencional do Linux é responsável por implementar reservas e conjuntos de reservas, além dos mecanismos para controle de admissão (decidir se uma reserva é aceita ou não), escalonamento de recursos (quando exatamente cada reserva é satisfeita), monitoração de uso (quanto de sua reserva cada programa usou) e imposição das reservas (não permitir que programas usem recursos além do permitido).

Em pontos apropriados do kernel do Linux são introduzidas chamadas para o módulo RK, o qual fornece uma interface apropriada e, por sua vez, utiliza funções do próprio Linux para controlar entidades do kernel, tais como processos e *device-drivers*. Chamadas para o módulo RK foram colocadas dentro da função de escalonamento "schedule()" do Linux, no atendimento de uma interrupção, no término do atendimento

das interrupções e na saída do kernel, quando a execução passa de modo kernel para modo usuário. Linux/RK também utiliza o recurso “Proc” do sistemas de arquivos para fornecer informações sobre o sistema, tais como situação das reservas e dos recursos.

Um temporizador de alta precisão é utilizado para garantir os limites das reservas aceitas. O relógio do hardware é sempre programado para gerar uma interrupção no próximo instante de interesse, evitando os erros associados com temporizadores que geram interrupções periódicas. Entretanto, como o kernel do Linux espera interrupções de timer periódicas, o módulo RK faz a propagação das mesmas para o kernel do Linux nos momentos apropriados.

O trabalho realizado no Linux/RK considerou basicamente a questão do compartilhamento do processador. Entretanto, em sistemas operacionais complexos, muitos outros recursos são disputados pelas tarefas. A pesquisa associada com o Linux/RK continua, e pretende integrar a reserva de tempo de acesso ao disco e tempo de acesso a rede. No momento que vários recursos são gerenciados, é necessário considerar o co-escalonamento de múltiplos recursos, isto é, considerar que o processo em questão pode precisar utilizar suas reservas dos vários recursos simultaneamente para realizar o seu trabalho.

O mecanismo de reservas é limitado para o caso de tarefas esporádicas, isto é, tarefas cujo momento de ativação não é conhecido, embora exista um intervalo mínimo de tempo entre ativações. Sistemas baseados em prioridades conseguem lidar naturalmente com este tipo de tarefa. A principal classe de aplicações para o Linux/RK são aquelas que lidam com audio e vídeo [RAJKUMAR 1998]. Essas aplicações possuem basicamente tarefas periódicas, o que torna esta limitação um problema menor.

3.3 RED-Linux

O propósito do RED-Linux (Real-Time and Embedded Linux), desenvolvido na Universidade da California em Irvine, é suportar aplicações embutidas através de um kernel eficiente e flexível [WANG 1999]. Em termos de eficiência, ele implementa temporização de alta resolução e mecanismos para reduzir a latência das interrupções. Em termos de flexibilidade, o projeto RED-Linux fornece suporte de escalonamento tempo real para o Linux, através da integração de escalonadores baseados em prioridade, baseados em compartilhamento de recursos e baseados na construção de escalas. O objetivo é suportar algoritmos de escalonamento dependentes da aplicação, os quais podem ser colecionados em uma biblioteca e reusados em outras aplicações.

RED-Linux implementa um mecanismo subjacente capaz de suportar os três paradigmas de escalonamento tempo real mais populares. No escalonamento baseado em prioridades, cada tarefa recebe uma prioridade e a tarefa com prioridade mais alta é sempre escolhida para execução. No escalonamento baseado no compartilhamento, cada tarefa de tempo real possui uma fatia dos recursos (um nível de utilização), e todas as tarefas compartilham os recursos respeitando esta divisão pré-estabelecida. No escalonamento baseado em escalas de execução, os instantes de tempo nos quais cada tarefa inicia, é suspensa, retoma sua execução e termina são previamente calculados e implementados através de uma escala de execução construida pelo escalonador.

Para suportar os três paradigmas, cada ativação de cada tarefa possui como atributos uma prioridade, um tempo de início, um tempo de fim e um orçamento. Os tempos de início e de fim definem juntos o intervalo de tempo potencial para execução

da ativação. A prioridade especifica a ordem relativa para execução das mesmas. O orçamento especifica o tempo total de execução permitido para cada ativação. Uma definição cuidadosa dessas quatro atributos para cada tarefa da aplicação permite a efetiva implementação de cada um dos três paradigmas de escalonamento citados antes, e também soluções resultantes da mistura de dois deles ou mesmo de todos os três.

O mecanismo implementado no RED-Linux é dividido em dois componentes: o Alocador e o Disparador. O Disparador implementa o mecanismo de escalonamento básico, respeitando os atributos de cada tarefa. O Alocador implementa a política que gerencia o tempo do processador e os recursos do sistema com o propósito de atender aos requisitos temporais das tarefas da aplicação, através da definição dos vários atributos. A política de escalonamento (Alocador) pode ser modificada sem alterar os mecanismos de escalonamento de baixo nível (Disparador).

O Disparador é implementado como um módulo do kernel. Ele é responsável por escalonar tarefas de tempo real que foram registradas com o Alocador, isto é, determinar a sua ordem de execução. Tarefas convencionais são escalonadas pelo escalonador original do Linux quando nenhuma tarefa de tempo real estiver pronta para executar ou durante o tempo reservado para tarefas convencionais pela política em uso.

O Alocador é utilizado para definir os atributos de escalonamento de cada nova tarefa tempo real. Na maioria das aplicações ele pode executar fora do kernel, como tarefa da aplicação ou um servidor auxiliar. Executando fora do kernel ele pode ser mais facilmente substituido pelo desenvolvedor da aplicação, se isto for necessário. O RED-Linux inclui uma API para que o Alocador possa interagir com o Disparador. Por exemplo, uma política de escalonamento adaptativa pode obter informações do Disparador e com isto alterar os atributos em vigor. Tarefas de tempo real inicialmente registram-se com o Alocador que, por sua vez, informa os atributos delas para o Disparador. O Alocador executa como a tarefa tempo real de mais alta prioridade e faz o mapeamento da política de escalonamento como selecionada pela aplicação para o mecanismo disponível no kernel do RED-Linux.

3.4 Real-Time Linux

O RT-Linux [YODAIKEN 1997] é uma extensão do Linux que se propõe a suportar tarefas com restrições temporais críticas. O seu desenvolvimento iniciou no "Department of Computer Science" do "New Mexico Institute of Technology". Atualmente o sistema é mantido principalmente pela empresa FSMLabs.

O RT-Linux é um sistema operacional no qual um microkernel de tempo real co-existe com o kernel do Linux. O objetivo deste arranjo é permitir que aplicações utilizem os serviços sofisticados e o bom comportamento no caso médio do Linux tradicional, ao mesmo tempo que permite tarefas de tempo real operarem sobre um ambiente mais previsível e com baixa latência. O microkernel de tempo real executa o kernel convencional como sua tarefa de mais baixa prioridade (Tarefa Linux), usando o conceito de máquina virtual para tornar o kernel convencional e todas as suas aplicações completamente interrompíveis.

Todas as interrupções são inicialmente tratadas pelo microkernel de tempo real, e são passadas para a Tarefa Linux somente quando não existem tarefas de tempo real para executar. Para minimizar mudanças no kernel convencional, o hardware que controla interrupções é emulado. Quando o kernel convencional "desabilita

interrupções", o software que emula o controlador de interrupções passa a enfileirar as interrupções que acontecerem e não forem completamente tratadas pelo microkernel de tempo real.

Tarefas de tempo real não podem usar as chamadas de sistema convencionais nem acessar as estruturas de dados do kernel Linux que podem gerar bloqueios. Tarefas de tempo real e tarefas convencionais podem comunicar-se através de filas sem bloqueio e memória compartilhada. As filas, chamadas de RT-FIFO, são na verdade *buffers* utilizados para a troca de mensagens, projetadas de tal forma que tarefas de tempo real nunca são bloqueadas.

Uma aplicação tempo real típica consiste de tarefas de tempo real incorporadas ao sistema na forma de módulos de kernel carregáveis. Em termos de espaço de endereçamento, tanto o microkernel como as tarefas de tempo real habitam o espaço de sistema do Linux. Aplicações também incluem tarefas Linux convencionais, as quais são responsáveis por funções tais como o registro de dados em arquivos, atualização da tela, comunicação via rede e outras funções sem restrições temporais. Um exemplo típico são aplicações de aquisição de dados. Observe que tanto as tarefas de tempo real como o próprio microkernel são carregadas como módulos adicionais ao kernel convencional.

Os serviços oferecidos pelo microkernel de tempo real são mínimos: tarefas com escalonamento baseado em prioridades fixas e alocação estática de memória. A comunicação entre tarefas de tempo real utiliza memória compartilhada e a sincronização pode ser feita via desabilitação das interrupções de hardware. Existem módulos de kernel opcionais que implementam outros serviços, tais como um escalonador EDF e implementação de semáforos. O mecanismo de módulos de kernel do Linux permite que novos serviços sejam disponibilizados para as tarefas de tempo real. Entretanto, quanto mais complexos estes serviços mais difícil será prever o comportamento das tarefas de tempo real. Ao prover apenas serviços mínimos, o microkernel do RT-Linux consegue excelente determinismo temporal, sendo capaz de suportar tarefas com períodos da ordem de dezenas de microsegundos em um microcomputador de mesa típico.

A descrição feita aqui refere-se a primeira versão do RT-Linux, a qual provia apenas o essencial para tarefas de tempo real. A segunda versão manteve o projeto básico mas aumentou o conjunto de serviços disponíveis para tarefas de tempo real, ao mesmo tempo que procurou fornecer uma interface Posix também a nível do microkernel. A inclusão do Posix deve-se principalmente à demanda de empresas que gostariam de portar aplicações existentes para este sistema, e a disponibilidade de uma interface Posix no microkernel torna este trabalho mais fácil.

O sistema RTAI é outra variação do Linux para tempo real que utiliza uma abordagem similar a do RT-Linux. Informações sobre o RTAI podem ser encontradas em “http://www.rtai.org”.

3.5 Monta Vista Linux

O “Linux kernel preemptability enhancement” é um projeto de código aberto cuja proposta é deixar o kernel do Linux completamente preemptável [WEINBERG 2001]. O projeto é patrocinado pela empresa Monta Vista Software, tendo sido incorporado ao kernel oficial do Linux a partir da versão v2.5.4.

O modelo de preempção usado é permitir que o kernel seja preemptado a qualquer momento, com algumas rápidas exceções. A preempção continua proibida quando um tratador de interrupção está executando, na realização de processamento associado com tratadores de interrupção que foram postergados ("*bottom half*"), quando a *thread* atual mantém a posse de mecanismos de sincronização do kernel tais como *spinlock*, *writelock* e *readlock* (nesses momentos o código do kernel não é reentrante) e, finalmente, quando o escalonador está executando. Nos demais momentos, e quando uma dessas condições deixa de ser verdadeira, o processo executando código do kernel pode ser preemptado.

A abordagem da Monta Vista concentra as alterações do código do kernel nos tratadores de interrupções, no funcionamento dos mecanismos de bloqueio usados dentro do kernel e na programação do escalonador (o algoritmo em si continua baseado em prioridades fixas).

No longo prazo, MontaVista investiga se os pontos onde preempção é proibida podem ser completamente eliminados, protegendo as seções críticas de curta duração com *spinlocks* que também desabilitam interrupções no processador local, e protegendo as seções críticas de longa duração com *mutexes* (aquelas maiores que dois chaveamentos de contexto). O *overhead* será reduzido pois não será mais necessário testar por preempção no final de um trecho do código onde a preempção estava proibida. A preempção ocorreria automaticamente como parte do serviço de atendimento de interrupções que liberam um processo de maior prioridade.

4. Conclusões

Na literatura de tempo real existe teoria suficiente para a construção de um sistema operacional capaz de atender com qualidade os requisitos temporais das aplicações. Entretanto, para que isto seja possível, são necessárias certas disciplinas de projeto, como descritas neste artigo. Embora seja difícil aplicar a teoria em sistemas operacionais antigos, construídos sem preocupações de tempo real, é possível adaptar sistemas operacionais existentes para melhorar o seu comportamento. Este artigo discutiu vários aspectos construtivos relevantes, ao mesmo tempo que descreveu várias adaptações existentes do Linux. É razoável esperar que, em alguns anos, existirão versões do Linux com capacidade de suportar com precisão tarefas de tempo real, ao mesmo tempo que disponibilizam todos os serviços usuais de um kernel completo.

Referências

[ABENI 2002] L. Abeni, A. Goel, C. Krasic, J. Snow, J. Walpole. A Measurement-Based Analysis of the Real-Time Performance of Linux. Proc. of the Real-Time Technology and Applications Symposium, 2002.

[ANGELOV 2002] C. K. Angelov, I. E. Ivanov, A. Burns. HARTEX - a safe real-time kernel for distributed computer control systems. Software - Practice and Experience 32(3): 209-232, 2002.

[AUDSLEY 1993] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. Software Engineering Journal, Vol. 8, No. 5, pp.284-292, 1993.

[AUDSLEY 2001] N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. Information Processing Letters, vol. 79, number 1, pages 39-44, 2001.

[BOVET 2001] D. P. Bovet, M. Cesati. Understanding the Linux Kernel. 2nd edition, O´Reilly, 2001.

[CARLSSON 2002] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, B. Lisper. Worst-Case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System. 2nd Work. on Real-Time Tools, Denmark, Aug. 2002.

[CAYSSIALS 1999] R. Cayssials, J. Orozco, J. Santos, R. Santos. Rate Monotonic scheduling of real-time control systems with the minimum number of priority levels. The 11th Euromicro Conf. on Real-Time Systems (ECRTS99), York, June 1999.

[EULER 1999] J. Euler, M. do C. Noronha, D. M. da Silva, Estudo de Caso: Desempenho do Sistema Operacional Linux para Aplicações Multimídia em Tempo Real. Anais do II Workshop de Tempo Real, Salvador-BA, 25-28 de maio de 1999.

[KATCHER 1993] D. Katcher, H. Arakawa, J. Strosnider. Engineering and Analysis of Fixed-Priority Schedulers. IEEE Trans. on Soft. Eng., Vol. 19, pp. 920-934, 1993.

[MEHNERT 2002] F. Mehnert, M. Hohmuth, H. Hartig. Cost and benefit of separate address spaces in real-time operating systems. 23rd IEEE Real-Time Syst. Symp., Texas, Dec. 2002.

[OIKAWA 1998] S. Oikawa, R. Rajkumar. Linux/RK: A portable resource kernel in Linux. IEEE Real-Time Systems Symposium, work-in-progress section, Madrid, December 1998.

[OIKAWA 1999] S. Oikawa, R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced behavior. IEEE Real-Time Technology and Applications Symposium, Vancouver, June 1999.

[OLIVEIRA 2003] R. de Oliveira. Aspectos Construtivos dos Sistemas Operacionais de Tempo Real. Workshop de Tempo Real, Natal-RN, maio 2003. Submetido.

[RAJKUMAR 1998] R. Rajkumar, K. Juvva, A. Molano, S. Oikawa. Resource kernels: a resource-centric approach to real-time systems. Proc. of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998.

[SRINIVASAN 1998] B. Srinivasan, S. Pather, R. Hill, F. Ansari, D. Niehaus. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. Proc. of the Real-Time Technology and App. Symp., june 1998.

[WANG 1999] Y.-C. Wang, K.-J. Lin. Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel. Proc. of the Real-Time Systems Symposium, December 1999.

[WEINBERG 2001] B. Weinberg, C. Lundholm. Embedded Linux – Ready for Real-Time. Third Real-Time Linux Workshop, Milano, Italy, Nov. 2001.

[YODAIKEN 1997] V. Yodaiken, M. Barabanov. A Real-Time Linux. Proc. of the Linux App. Development and Deployment Conference (USELINUX), Anaheim, CA, Jan. 1997.