Aspectos Construtivos dos Sistemas Operacionais de Tempo Real

Rômulo Silva de Oliveira

romulo@das.ufsc.br
LCMI-DAS - Universidade Federal de Santa Catarina
Caixa Postal 476, Florianópolis-SC, 88040-900

Resumo

Aplicações de tempo real são mais facilmente construídas se puderem aproveitar os serviços de um sistema operacional. Apesar da teoria de escalonamento tempo real ter tido um grande avanço nos últimos 20 anos, ainda não é corrente o seu uso na modelagem e análise de sistemas operacionais complexos. Este artigo analisa aspectos construtivos de sistemas operacionais e suas implicações sob a ótica da teoria de escalonamento tempo real baseada em prioridades fixas. Busca-se determinar como os mecanismos empregados podem ser modelados, ou quais as dificuldades que eles apresentam para a modelagem.

Palavras-chave: tempo real, sistema operacional, escalonamento.

1. Introdução

Assim como aplicações convencionais, aplicações de tempo real são mais facilmente construídas se puderem aproveitar os serviços de um sistema operacional (SO). Desta forma, o programador da aplicação não precisa preocupar-se com a gerência dos recursos básicos (processador, memória física, controladores de periféricos). Ele utiliza as abstrações de mais alto nível criadas pelo sistema operacional (tarefas, segmentos, arquivos).

Uma vez que tanto a aplicação como o 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 20 anos. É razoável afirmar que 10 anos atrás já existia um ferramental teórico capaz de permitir a modelagem e análise de escalonabilidade de muitos sistemas de software com requisitos de tempo real. Por exemplo, a teoria considerada neste artigo já estava estabelecida em 1993.

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 como os mecanismos empregados podem ser modelados, ou quais as dificuldades que eles apresentam para a modelagem. Ao mesmo tempo, são feitas indicações sobre quais mecanismos são mais apropriados para a construção de um sistema operacional que deverá ter seu comportamento temporal modelado e analisado.

A teoria de tempo real é extensa. No sentido de obter-se resultados mais concretos, este trabalho emprega especificamente a análise de tempo de resposta de sistemas baseados em prioridade fixa, conforme descrito na seção 2. São analisados aspectos construtivos de sistemas operacionais e suas implicações sob a ótica desta teoria de escalonamento.

Algumas questões de modelagem discutidas aqui apareceram antes na literatura [1]. Pesquisas recentes focaram alguns dos aspectos construtivos abordados neste artigo, sem porém utilizar os modelos da teoria de escalonamento baseada em prioridade fixa. Em [2] são discutidos temporizadores e preempção. Em [3] é considerado o problema da determinação do tempo de execução no pior caso do código encontrado em sistemas operacionais. Em [4] é analisado o custo de manter espaços de endereçamento separados.

O propósito deste trabalho é colaborar para que sistemas operacionais com funcionalidade ampla possam ser construidos de forma a permitir uma modelagem e análise de pior caso (previsibilidade determinista). A seção 2 rapidamente descreve a teoria de escalonamento considerada. A seção 3 discute vários aspectos construtivos de sistemas operacionais sob a luz da teoria de escalonamento. A seção 4 apresenta as conclusões.

2. Análise de Escalonabilidade para Sistemas de Prioridade Fixa

Neste trabalho são considerados os métodos de análise de escalonabilidade desenvolvidos por Audsley em sua tese [5], para tarefas com deadline menor ou igual ao período, os quais estendem o trabalho anterior de Joseph e Pandya [6]. Posteriormente, Tindell em sua tese [7] estendeu este tipo de análise para tarefas com deadline arbitrário, entre outras coisas. Descrições da abordagem podem ser encontradas em [1], [8], [9], [10]. Embora esta abordagem também tenha sido estendida para ambientes distribuídos, este artigo trata exclusivamente de sistemas com um único processador.

De forma simplificada, a análise de escalonabilidade em questão assume que uma prioridade fixa é associada a cada tarefa em tempo de projeto. Durante a execução é utilizado um escalonador preemptivo baseado em prioridades. Testes de escalonabilidade em tempo de projeto calculam o tempo máximo de resposta para cada tarefa e os comparam com os respectivos deadlines, avaliando dessa forma a escalonabilidade do sistema. Os testes usados verificam um conjunto de tarefas com prioridades fixas arbitrárias, não necessariamente atribuídas através do deadline monotônico. Embora, em muitos casos, a atribuição através de deadline monotônico seja ótima [11].

No modelo suportado, as tarefas podem ser periódicas ou esporádicas com um intervalo de tempo mínimo entre ativações. Podem existir seções críticas que geram relações de exclusão mútua entre as tarefas. O modelo também admite a existência de *release jitter* na liberação das tarefas e relações de precedência entre elas.

A limitação de espaço impede uma descrição detalhada das equações empregadas na análise de escalonabilidade usada neste trabalho. A título de ilustração, considere um sistema formado por N tarefas, onde cada tarefa Ti é periódica com período Pi ou esporádica com intervalo mínimo entre ativações Pi. Cada tarefa Ti possui uma prioridade fixa $\rho(T_i)$. Cada tarefa Ti também possui um tempo máximo de computação Ci, um deadline Di e um release jitter máximo Ji. É suposto que Di≤Pi para todas as tarefas. Em cada ativação da tarefa Ti ela pode ser bloqueada por tarefas de menor prioridade no máximo Bi unidades de tempo.

Dada as condições acima, o tempo máximo de resposta Ri de cada tarefa Ti pode ser calculado através das equações abaixo, onde Wi é obtido através de um processo iterativo, com complexidade pseudo-polinomial. O conjunto HP(i) é formado por todas as tarefas com prioridade maior ou igual a prioridade da tarefa Ti.

$$W_i = C_i + B_i + \sum_{j \in HP(i)} \left\lceil \frac{J_j + W_i}{P_j} \right\rceil \times C_j \qquad\qquad R_i = J_i + W_i$$

As equações apresentadas continuam válidas quando políticas de gerência de recursos tais como herança de prioridade e *priority ceiling* são usadas nas situações de exclusão mútua.

Por outro lado, as equações apresentadas são válidas apenas para deadline menor ou igual ao período. Equações semelhantes são apresentadas em [9] para deadlines arbitrários.

3. 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, como podem ser modeladas as contribuições do sistema operacional para o tempo de resposta das tarefas da aplicação.

3.1 Algoritmo de Escalonamento Apropriado

Ao considerar como o sistema operacional pode afetar o comportamento da aplicação no tempo, o primeiro aspecto sempre lembrado é o algoritmo de escalonamento usado. 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 deste trabalho, para aplicar a teoria descrita na seção 2, o SO deve oferecer prioridades preemptivas. Isto não é um problema, pois este é o algoritmo implementado pela maioria dos sistemas operacionais de tempo real. Basta que cada tarefa no sistema tenha uma prioridade fixa, tanto as tarefas da aplicação como as do sistema.

3.2 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 [12] 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 [12] 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 [13] 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. Se isto não for verdade, então as equações da seção 2 precisam ser ajustadas, e a escalonabilidade do sistema pode ficar comprometida.

3.3 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 também incluem mecanismos que reduzem automaticamente a prioridade de uma *thread* na medida que ela consome tempo de processador. Este tipo de 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 SOPG, 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.

3.4 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, apresentando as propriedades tempo máximo de execução e período ou intervalo mínimo entre ativações.

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.

3.5 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 associada com 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.

3.6 *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. Do ponto de vista da teoria de tempo real descrita na seção 2, é importante que toda e qualquer computação

esteja associada com uma tarefa (ou pseudo-tarefa) de prioridade fixa.

3.7 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 [14]. Em [10] é mostrado experimentalmente que a liberação simultânea de 20 tarefas periódicas pode ocupar até 570us do tempo de um processador M68020 executando o sistema operacional Olympus. Nesse mesmo processador e sistema operacional a liberação de uma única tarefa demora apenas 76us.

Segundo [15], 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.

Em termos de modelagem, a passagem de uma tarefa de uma fila de espera para a fila do processador corresponde a uma computação, realizada por um tratador de interrupção ou por uma *thread* de kernel. Observe que esta ação pode ou não estar associada com um subseqüente chaveamento de contexto, conforme a prioridade da tarefa liberada.

3.8 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.

3.9 Temporizadores de Alta Resolução

Sistemas operacionais em geral oferecem temporizadores para as tarefas. Tarefas da aplicação armam temporizações ao final das quais uma determinada ação ocorre. Essa ação pode ser assíncrona (envio de um sinal Unix) ou síncrona (liberação da tarefa após um *sleep*). Temporizadores são utilizados na implementação de *time-out*, *watch-dog*, 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 apenas de 10ms em 10ms. Ao final, a espera de 15ms acabou consumindo 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.

3.10 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.

Em termos de modelagem, 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 será calcular este tempo. Projetos futuros de SOTR terão que minimizar este problema através de prática construtivas que reduzam a variância nos tempos de execução das chamadas de sistema.

3.11 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. Em termos de modelagem isto corresponde a um bloqueio.

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 o próximo ponto de preempção. Obviamente, um SOTR deve ser preemptivo, ainda que em determinados momentos interrupções precisem ser desabilitadas em função das estruturas de dados acessadas.

3.12 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. Para efeitos de análise, tais mecanismos reduzem a inversão de prioridades dentro do kernel, reduzindo o tempo de bloqueio sofrido por tarefas que fazem chamadas de sistema.

3.13 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.

3.14 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. Entretanto, 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. Em termos de modelagem, 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 do sistema, pois sua duração está associada com toda a fila de tarefas que existia quando a tarefa de alta prioridade solicitou o recurso.

3.15 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.

4. Conclusões

Na literatura de tempo real existe teoria suficiente para a modelagem completa de um sistema operacional. Entretanto, para que isto seja possível, são necessárias certas disciplinas de projeto, como descritas neste artigo. Embora seja difícil aplicar a modelagem em sistemas operacionais antigos, construídos sem preocupações de tempo real, é viável aplicá-la a um sistema operacional completo, desde que ele seja construido com algum cuidado.

É razoável esperar que, em alguns anos, existirão sistemas operacionais completos (com funcionalidade semelhante ao Linux e ao Windows 2000), onde equações permitirão o cálculo do tempo máximo de resposta para cada tarefa do sistema, mesmo que tarefas utilizem livremente os serviços do sistema operacional.

Referências

[1] 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.

[2] 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.

[3] 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 Workshop on Real-Time Tools (RTTOOLS 2002), Denmark, Aug. 2002.

[4] F. Mehnert, M. Hohmuth, H. Hartig. Cost and benefit of separate address spaces in real-time operating systems. 23rd IEEE Real-Time Systems Symposium, Texas, dec. 2002.

[5] N. C. Audsley. Flexible Scheduling of Hard Real-Time Systems. Department of Computer Science Thesis, University of York, 1994.

[6] M. Joseph, P. Pandya. Finding Response Times in a Real-Time System. BCS Computer Journal Vol 29, N^{o} 5, pp. 390-395, 1989.

[7] K. W. Tindell. Fixed Priority Scheduling of Hard Real-Time Systems. Department of Computer Science Thesis, University of York, 1994.

[8] N. C. Audsley, A. Burns, A. J. Wellings. Deadline Monotonic Scheduling Theory and Application. Control Engineering Practice, Vol.1, No.1, pp. 71-78, february 1993.

[9] K. W. Tindell, A. Burns, A. J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. Real-Time Systems, pp. 133-151, 1994.

[10] A. Burns, K. Tindell, A. Wellings. Effective Analysis for Engineering Real-Time Fixed-Priority Schedulers. IEEE Trans. on Soft. Eng., Vol. 21, pp. 475-480, 1995.

[11] J. Y. T. Leung, J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. Performance Evaluation, 2 (4), pp. 237-250, dec. 1982.

[12] 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 Conference on Real-Time Systems (ECRTS99), York, England, June 9-11, 1999.

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

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

[15] 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.