POLÍTICA DE ESCALONAMENTO DE PROCESSOS EM LINUX

MARCOS VINÍCIUS DE MEIRA

Faculdade Integrado de Campo Mourão – Av. Irmãos Pereira, 670 - Cep: 87300-010
Campo Mourão – Paraná – Brasil. E-mail: mvmeira@grupointegrado.br

RESUMO

O problema básico de escalonamento em sistemas operacionais é como satisfazer simultaneamente objetivos conflitantes: tempo de resposta rápido, bom throughput para processos background (execução sem interação do usuário), evitar starvation, conciliar processos de alta prioridade com de baixa prioridade. O conjunto de regras utilizado para determinar como, quando e qual processo deverá ser executado é conhecido como política de escalonamento. Este artigo discute como o Linux implementa seu escalonador e qual a política empregada para determinar quais processos recebem o processador.

Palavras-Chave: *Linux, Escalonamento, Processos*

POLITIC OF PROCESSES SCHEDULING IN LINUX

ABSTRACT

The basic problem of scheduling in operational systems is as to satisfy objective simultaneously conflicting: fast, good time of reply throughput for processes background, to prevent starvation, to conciliate processes of high priority with of low priority. The set of rules used to determine as, when and which process will have to be executed is known as scheduling politics. This article argues as the Linux implements its scheduling and which the used politics to determine which processes receives the processor.

Key-Words: *Linux, Scheduling, Processes*

INTRODUÇÃO

O problema básico de escalonamento em sistemas operacionais é como satisfazer simultaneamente objetivos conflitantes: tempo de resposta rápido, bom *throughput* para processos *background*, evitar *starvation*, conciliar processos de alta prioridade com de baixa prioridade. O conjunto de regras utilizado para determinar como, quando e qual processo deverá ser executado é conhecido como política de escalonamento. Tradicionalmente, os processos são divididos em três grandes classes: processos interativos, processos *batch* e processos de tempo real. Em cada classe, os processos podem ser ainda subdivididos em *I/O bound* ou *CPU bound* de acordo com a proporção de tempo que ficam esperando por operações de entrada e saída ou utilizando o processador. O escalonador do Linux não distingue processos interativos de processos *batch*, diferenciando-os apenas dos processos tempo real. Como todos os outros escalonadores UNIX, o escalonador Linux privilegia os processos *I/O bound* em relação aos *CPU bound* de forma a oferecer um melhor tempo de resposta às aplicações interativas. O escalonador do Linux é baseado em *time-sharing*, ou seja, o tempo de processamento é dividido em fatias de tempo (*quantum*) as quais são alocadas aos processos. Se, durante a execução de um processo o *quantum* é esgotado, um novo processo é selecionado para execução, provocando então uma troca de contexto. Esse procedimento é completamente transparente ao processo e baseia-se em interrupções de tempo. Esse comportamento confere ao Linux em escalonamento do tipo *preemptivo*. O algoritmo de escalonamento do Linux divide o tempo de processamento em épocas. Cada processo, no momento de sua criação, recebe um *quantum* calculado no início de sua época. Outra característica do escalonador Linux é a existência de prioridades dinâmicas. O escalonador do Linux monitora o comportamento de um processo e ajusta dinamicamente sua prioridade. Este artigo discute como o Linux implementa seu escalonador e qual a política empregada para determinar quais processos recebem o processador.

GERENCIAMENTO DE PROCESSO

O subsistema de gerenciamento de processo é essencial para fornecer multiprogramação eficiente no Linux. Embora primordialmente responsável por alocar processos a processadores, o subsistema de gerenciamento de processo também deve entregar sinais, carregar módulos de núcleo e receber interrupções. O subsistema de gerenciamento de processo contém o escalonador de processo que permite que processos tenham acesso a um processador em tempo razoável.

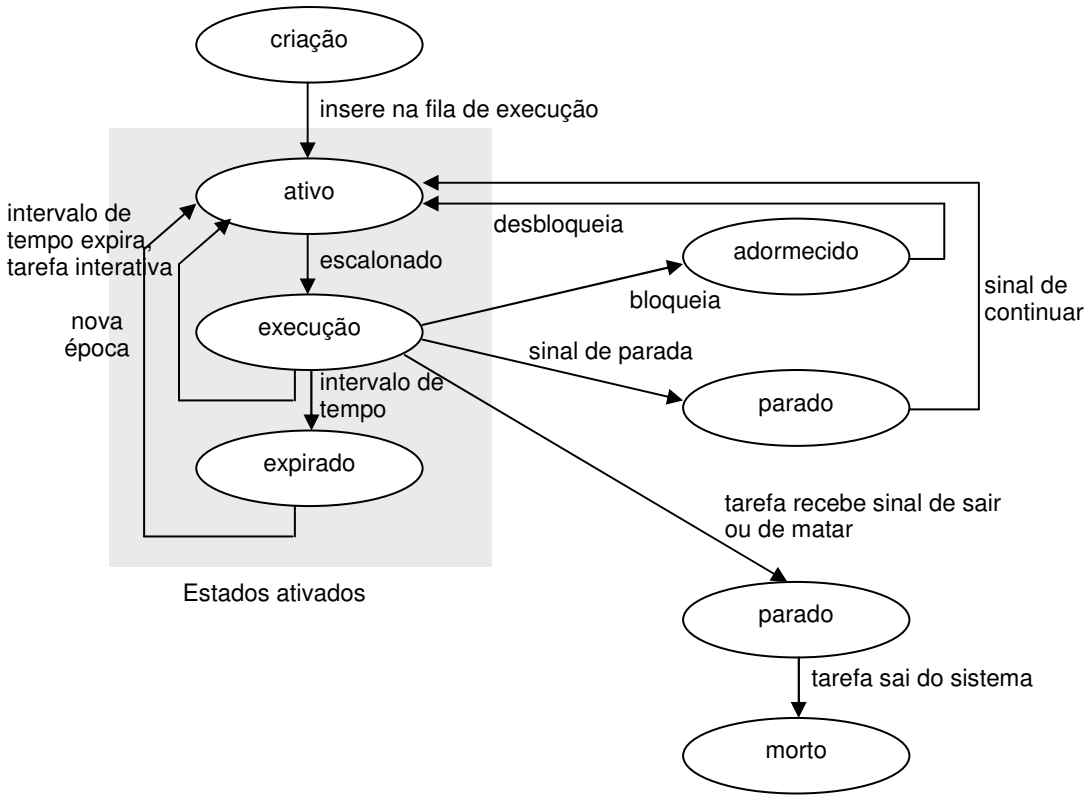
ORGANIZAÇÃO DE PROCESSOS E *THREADS*

Em sistemas Linux, ambos, processos e *theads*, são denominados tarefas, internamente são representados por uma única estrutura de dados. O gerenciador de processos mantém uma lista de todas as tarefas usando duas estruturas de dados. A primeira é uma lista circular, duplamente encadeada, na qual cada entrada contém ponteiros para as tarefas anteriores e posteriores da lista. Essa estrutura é acessada quando o núcleo tem de examinar todas as tarefas do sistema. A segunda é uma tabela *hash*. Quando uma tarefa é criada, recebe um identificador de processo (*Process Identifier* – PID) exclusivo. Identificadores de processos são passados para uma função *hash* para determinar sua localização na tabela de processos. O método de *hash* fornece acesso rápido à estrutura de dados de uma tarefa específica quando o núcleo conhece o seu PID. (Aivazian, 2001)

Cada tarefa da tabela de processo é representada por uma estrutura *task_struct* que serve como descritor de processo ou PCB. A estrutura *task_struct* armazena variáveis e estruturas aninhadas que contém informações que descrevem um processo.

Uma tarefa transita para o estado de execução quando é enviada para o processador, conforme figura 1 (Deitel, 2005). Uma tarefa entra no estado adormecido quando bloqueia, e no estado parado quando é suspensa. O estado zumbi indica que uma tarefa foi terminada, mas não eliminada do sistema. Ativo e expirado são estados de escalonamento de processo que não são armazenados na variável *state* (estado).

Figura 1. Diagrama de transição de estado da tarefa (Deitel, 2005)



Outras importantes variáveis específicas de tarefas permitem que o escalonador determine quando uma tarefa deve executar em um processador. Entre essas variáveis estão, as prioridades da tarefa, se a tarefa é de tempo real ou não e, se for, qual algoritmo de escalonamento de tempo real deve ser usado. (Rusling, 1999)

Estruturas aninhadas dentro de uma *task_struct* armazenam informações adicionais sobre uma tarefa. Uma estrutura desse tipo, denominada *mm_struct*, descreve a memória alocada a uma tarefa. Estruturas adicionais aninhadas dentro de uma *task_struct* contêm informações como valores de registradores que armazenam o contexto de execução de uma tarefa, controladores de sinal e os direitos de acesso da tarefa.(SchlapBach, 2002) Essas estruturas são acessadas por vários subsistemas de núcleo, além do gerenciador de processo.

Quando o núcleo é inicializado, normalmente carrega um processo denominado *init* que então usa o núcleo para criar todas as outras tarefas. (Aivazian, 2001) Tarefas são criadas por meio da chamada ao sistema *clone*, quaisquer chamadas a *fork* ou *vfork* são convertidas a chamadas ao sistema *clone* na hora da compilação. O propósito de *fork* é criar uma tarefa-filho cujo espaço de memória virtual é alocado usando *copy-on-write* para aprimorar o desempenho. Quando o processo-filho ou o processo-pai tentam escrever para uma página na memória, uma cópia da página é alocada ao escritor. A *copy-on-write* pode levar o mau desempenho se um processo chamar *execv* para carregar um novo programa imediatamente após a *fork*. O linux suporta a chamada *vfork* que melhora o desempenho quando processos-filho chamarem *execve*. A chamada *vfork* suspende o processo-pai até que o filho chame *execve* ou *exit* para garantir que o filho carregue suas novas páginas antes que o pai cause quaisquer operações dispendiosas de *copy-on-write*. A chamada *vfork* melhora ainda mais o desempenho por não copiar as tabelas de páginas do pai para o filho, porque são criadas entradas de tabela de página quando o filho chamar *execve*.

THREADS LINUX E A CHAMADA AO SISTEMA *CLONE*

O Linux fornece suporte para *threads* usando a chamada ao sistema *clone*, que habilita o processo que está chamando a especificar se o *thread* compartilha a memória virtual do processo, informações de sistema de arquivo, descritores de arquivo e controladores de sinais.(Walton, 1997)

A implementação de *threads* do Linux gerou muita discussão no que diz respeito à definição de um *thread*. Embora *clone* crie *threads*, estes não estão necessariamente de acordo com a especificação de *threads* POSIX (interface portável de sistema operacional). Por exemplo, dois ou mais *threads* criados por meio de uma chamada *clone* que especifica máximo compartilhamento de recursos, ainda mantêm diversas estruturas de dados que não são compartilhadas com todos os *threads* do processo, como direitos de acesso.(McCracken, 2002)

Quando *clone* é chamado a partir de um processo de núcleo (um processo que execute código de núcleo), ela cria um *thread* de núcleo que é diferente de outros *threads* no sentido de que acessa diretamente o espaço de endereçamento do núcleo. Diversos *daemons* (processos que executam periodicamente para realizar serviços de sistema) dentro do núcleo são implementados como *threads* de núcleo, esses *daemons* são serviços que ficam adormecidos até que sejam despertados para realizar tarefas, como descarregar páginas para o armazenamento secundário e escalonar interrupções de *software*. (Arcomano, 2002) Essas tarefas em geral estão relacionadas com manutenção e executam periodicamente.

Há diversos benefícios na implementação de *threads* Linux. Por exemplo, *threads* Linux simplificam código de núcleo e reduzem sobrecarga requisitando somente uma única cópia das estruturas de dados de gerenciamento de tarefas.(Walton, 2002) Além disso, embora *threads* Linux sejam menos portáveis do que *threads* POSIX, permitem aos programadores flexibilidade para controlar rigorosamente recursos compartilhados entre tarefas.(Drepper, 2002)

ESCALONAMENTO DE PROCESSO

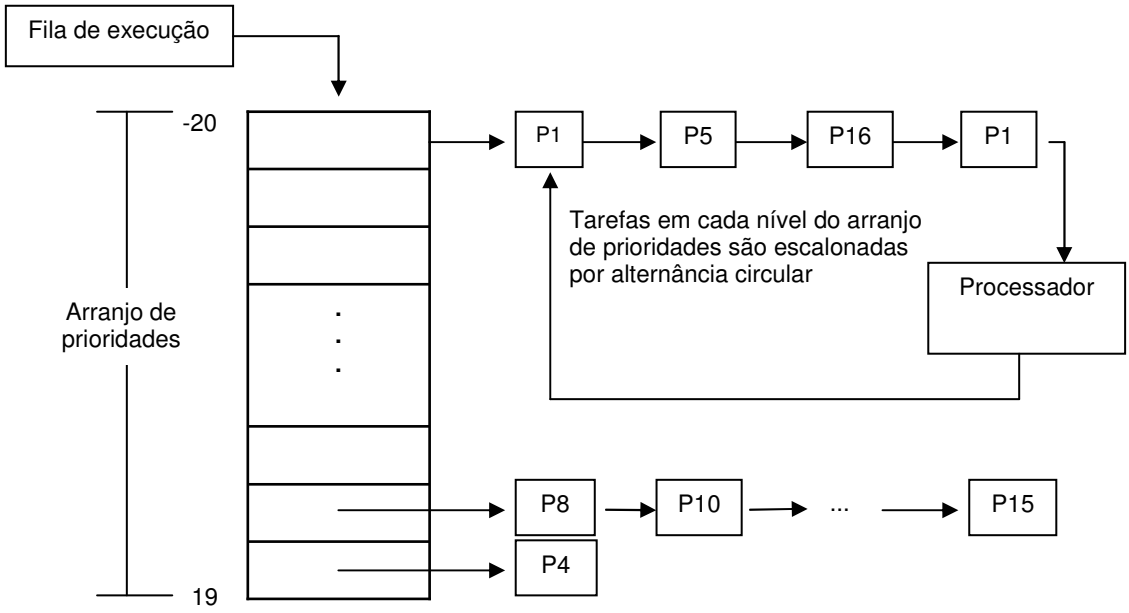
A meta do escalonador de processos Linux é executar todas as tarefas em uma quantidade razoável de tempo e, simultaneamente respeitar prioridades de tarefas, manter alta utilização e rendimento de recursos e reduzir a sobrecarga de operações de escalonamento. O escalonador de processo também aborda o papel do Linux no mercado de sistemas avançados de computador, escalando para arquiteturas SMP (multiprocessamento simétrico) e NUMA (acesso não uniforme à memória) e fornecendo, ao mesmo tempo, alta afinidade de processador. Um dos aprimoramentos mais significativos da

escalabilidade da versão 2.6 é que todas as versões de escalonamento são operações de tempo constante, o que significa que o tempo requerido para executar funções de escalonamento não depende do número de tarefas do sistema.(Molnar, 2003)
A cada interrupção do temporizador do sistema, o núcleo atualiza várias estruturas de dados de contabilidade e realiza operações de escalonamento conforme necessário. Porque o escalonador é preemptivo, cada tarefa executa até que seu *quantum* ou intervalo de tempo expire, um processo de prioridade mais alta torne-se executável ou o processo bloqueie. O intervalo de tempo de cada tarefa é calculado como uma função da prioridade do processo quando há uma liberação do processador. Para evitar que intervalos de tempo sejam demasiamente pequenos a ponto de impedir trabalho produtivo, ou tão grandes que aumentem os tempos de resposta, o escalonador assegura que o intervalo de tempo designado para cada tarefa esteja entre 10 e 200 intervalos de tempo. Quando uma tarefa sofre preempção, o escalonador salva o estado da tarefa em sua estrutura *task_struct*. Se o intervalo de tempo do processo expirar, o escalonador recalcula a prioridade do processo, determina o próximo intervalo de tempo de tarefa e seleciona o processo seguinte.

FILAS DE EXECUÇÃO

Uma vez criada uma tarefa por meio de *clone*, ela é colocada na fila de execução de um processador que contém referências a todas as outras tarefas que competem por execução naquele processador. Filas de execução, semelhantes a filas multiníveis de retorno, designam níveis de prioridades às tarefas. O arranjo de prioridades mantém ponteiros para cada nível da fila de execução. Cada entrada do arranjo de prioridades aponta para uma lista de tarefas, uma de prioridade i é colocada na i-ésima entrada de um arranjo de prioridades da fila de execução, como visto na figura 2 (Deitel, 2005). O escalonador seleciona a tarefa que está à frente da lista no nível mais alto do arranjo de prioridades. Se houver mais de uma tarefa em um nível do arranjo de prioridades, as tarefas serão selecionadas por meio de alternância circular. Quando uma tarefa entra no estado bloqueado ou adormecido (de espera) ou por qualquer outra razão é incapaz de executar, ela é retirada da sua fila de execução.

Figura 2. Arranjo de prioridades do escalonador. (Deitel, 2005)



Uma meta do escalonador é impedir adiamento, estabelecendo um período de tempo denominado época durante o qual cada tarefa da fila de execução executará no mínimo uma vez. Para distinguir processos considerados para tempo de processador daqueles que devem esperar até a próxima época, o escalonador define um estado ativo e um estado expirado. O escalonador seleciona somente processos que estão no estado ativo.

A duração de uma época é determinada pelo limite de inanição, um valor derivado empiricamente que fornece tarefas de alta prioridade com bons tempos de resposta e, ao mesmo tempo, assegura que tarefas de baixa prioridade sejam selecionadas com freqüência suficiente para realizar trabalho produtivo em uma quantidade de tempo razoável. Por default, o limite de inanição é estabelecido em 10n segundos, onde n é o número de tarefas da fila de execução. Quando a época corrente durou mais do que o limite de inanição, o escalonador transita cada tarefa ativa da fila de execução para o estado expirado, permitindo que tarefas de baixa prioridade executem. Quando todas as tarefas da fila de execução tiverem executado pelo menos uma vez, todas estarão no estado expirado. Nesse ponto, o escalonador transita todas as tarefas da fila de execução para o estado ativo e começa uma nova época.(Cross-referecing Linux)

Para simplificar a transição do estado expirado para o estado ativo, ao final de uma época o escalonador do Linux mantém dois arranjos de prioridades para cada processador. O arranjo de prioridades que contém tarefas no estado ativo é denominado lista ativa. O arranjo de prioridades que armazena tarefas expiradas é denominado lista inativa ou lista expirada. Quando uma tarefa transita do estado ativo para o estado expirado, é colocada no arranjo de prioridades da lista expirada no nível de prioridade que tinha quando transitou para o estado expirado. Ao final de uma época, todas as tarefas estão localizadas no estado expirado e devem transitar para o estado ativo. O escalonador executa essa operação rapidamente apenas trocando (*swapping*) os ponteiros da lista expirada e da lista ativa. Por manter dois arranjos de prioridades por processo, o escalonador pode transitar todas as tarefas de uma fila de execução usando uma única operação de troca, um aprimoramento do desempenho que, em geral, compensa a sobrecarga nominal de memória devido a essa operação.(Cross-referecing Linux)

O escalonador Linux escala para sistemas de multiprocessadores mantendo uma fila de execução para cada processador físico do sistema. Uma razão para filas de execução por processador é designar a execução de tarefas em determinados processadores para explorar a afinidade de processador. Em algumas arquiteturas de processadores, processos atingem desempenho mais alto quando os dados de uma tarefa estão armazenados na *cache* do processador. Consequentemente, tarefas podem atingir desempenho mais alto se forem consistentemente designada para um único processador. Contudo, filas de execução por processador correm o risco de desbalancear cargas de processador resultando em redução do desempenho e do rendimento do sistema.

ESCALONAMENTO POR PRIORIDADE

No escalonador do Linux, a prioridade de uma tarefa afeta o tamanho de seu período de tempo e a ordem em que ele executa em um processador. Quando são criadas, as tarefas recebem uma prioridade estática, também denominada valor bom. O escalonador reconhece 40 níveis distintos de prioridades que vão de -20 a 19. Seguindo a convenção UNIX, valores menores indicam prioridades mais altas no algoritmo de escalonamento (ou seja, -20 é a prioridade mais alta que um processo pode atingir).

Uma meta do escalonador do Linux é fornecer um alto nível de interatividade no sistema. Porque tarefas interativas comumente bloqueiam para executar E/S ou dormir, o escalonador eleva dinamicamente a prioridade de uma tarefa (decrementando o valor da prioridade estática) que entregar seu processador antes de seu intervalo de seu intervalo de tempo expirar. Isso é aceitável, pois processos *I/O bound* normalmente usam processadores apenas brevemente antes de gerar uma requisição de E/S. Assim atribuir prioridades altas a tarefas *I/O bound* tem pouco efeito sobre tarefas *CPU bound*, que poderiam utilizar o processador durante horas por vez caso o sistema o disponibilizasse em base não preemptiva. O nível de prioridade modificado é denominado prioridade efetiva de uma tarefa, a qual é calculada quando uma tarefa adormece ou consome seu intervalo de tempo. A prioridade efetiva de uma tarefa determina o nível do arranjo de prioridades no qual ela é colocada. Por conseguinte, a tarefa cuja prioridade for promovida é colocada em um nível mais baixo do arranjo de prioridades, o que significa que executará antes de tarefas que têm valores de prioridades efetivas mais altos.

Para melhorar ainda mais a interatividade, o escalonador penaliza uma tarefa *CPU bound* aumentado o valor de sua prioridade estática, o que coloca uma tarefa *CPU bound* em um nível mais alto do arranjo de prioridades, ou seja, tarefas de prioridades efetivas menores serão executadas antes dela. Novamente, em última análise, isso tem pouco efeito sobre tarefas *CPU bound* porque tarefas interativas de prioridades mais altas executam apenas brevemente antes de bloquear.

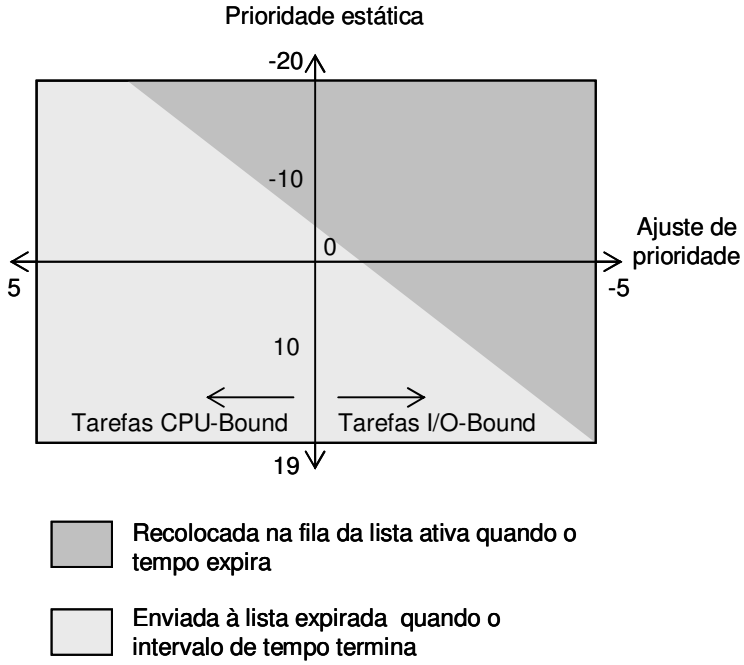
Para garantir que uma tarefa execute na prioridade que lhe foi designada inicialmente, ou próximo dela, o escalonador de tarefas não permite que a diferença entre a prioridade efetiva de uma tarefa e sua prioridade estática seja maior do que cinco unidades. Nesse sentido, o escalonador honra os níveis de prioridade designados a uma tarefa quando ela foi criada.

OPERAÇÕES DE ESCALONAMENTO

O escalonador elimina uma tarefa de um processador se ela for interrompida, sofrer preempção ou bloquear. Toda vez que uma tarefa é eliminada de um processador, o escalonador calcula um novo período de tempo. Se a tarefa bloquear, ou se for incapaz de executar por qualquer outra razão, ela será desativada, o que significa que será retirada da fila de execução até que fique pronta pra executar. Caso contrário, o escalonador determinará se a tarefa deve ser colocada na lista ativa ou na lista inativa. O algoritmo que determina isso foi derivado empiricamente para proporcionar bom desempenho, seus fatores primários são as prioridades estáticas e efetivas de uma tarefa.

O resultado do algoritmo é apresentado na Figura 3 (Linux kernel source code). O eixo y da figura indica valor da prioridade de uma tarefa. A região sombreada indica conjuntos de valores de prioridades estáticas e ajustes de prioridade que fazem com que uma tarefa seja reescalonada, significando que ela é colocada no final de seu arranjo de prioridades correspondente na lista ativa. Em geral, se uma tarefa for de alta prioridade ou se sua prioridade efetiva recebeu um bônus significativo, será reescalonada, o que permite que tarefas *I/O bound* e tarefas interativas executem mais do que uma vez por época. Na região não sombreada, as tarefas que têm baixa prioridade ou receberam penalidades de prioridades são colocadas na lista expirada.

Figura 3. Valores de prioridades e seus níveis de interatividade correspondentes (Linux k.s.c.)



Quando um processo usuário executa *clone*, pode parecer alocar a cada filho seu próprio intervalo de tempo. Todavia, se uma tarefa gerar um grande número de tarefas e todos os seus filhos receberem seus próprios intervalos de tempo, outras tarefas do sistema poderão sofrer tempos de resposta ruins durante aquela época. Para melhorar a distribuição, o Linux requer que cada processo-pai compartilhe inicialmente seu intervalo de tempo com seus filhos quando um processo executar *clones*. O escalonador impõe esse requisito designado metade do intervalo de tempo original do pai para ambos, o processo-pai e seu filho, que também gera filhos. Para evitar que processos legítimos recebam baixos níveis de serviços devido a filhos que geram processos, esse intervalo de tempo reduzido

aplica-se somente durante o restante da época na qual o filho foi gerado.

ESCALONAMENTO DE MULTIPROCESSADOR

Pelo fato do escalonador de processos manter as tarefas em uma fila de execução por processo, estas em geral exibirão alta afinidade de processador. Isso significa que uma tarefa provavelmente será selecionada para o mesmo processador em cada um de seus intervalos de tempo, o que pode aumentar o desempenho quando os dados e instruções de uma tarefa estiverem localizados nos *caches* de um processador. Entretanto, um esquema desse tipo poderia permitir que um ou diversos processadores de um sistema ficassem ociosos até mesmo durante um período de carga pesada do sistema. Para evitar tal situação, se o escalonador detectar que um processador esteja ocioso, executará balanceamento de carga para migrar tarefas de um processador para outro de modo que melhore a utilização de recursos. Se o sistema tiver apenas um processador, as rotinas de balanceamento são eliminadas do núcleo quando este é compilado.

O escalonador determina se o núcleo deve realizar rotinas de balanceamento de carga após cada interrupção do temporizador. Se o processador que emitiu a interrupção do temporizador estiver ocioso, o escalonador tentará migrar tarefas do processador que tiver carga mais pesada, ou seja, o que tiver o maior número de processos em sua fila de execução, para o processador ocioso. Para reduzir a sobrecarga de balanceamento de carga, se o processador que acionou a interrupção não estiver ocioso, o escalonador tentará transferir tarefas para aquele processador a cada 200 interrupções do tempo em vez de após cada interrupção do temporizador. (Linux kernel source code)

O escalonador determina a carga do processador usando o tamanho médio de cada fila de execução nas últimas várias interrupções de temporizador para minimizar o efeito de variações de cargas do processador sobre o algoritmo de balanceamento de carga. Pelo fato de as cargas de processadores tenderem a mudar rapidamente, a meta do balanceamento de carga não é ajustar o tamanho de duas filas de execução até que fiquem com o mesmo tamanho, mas reduzir o desequilíbrio entre o número de tarefas de cada fila de execução. O objetivo é que tarefas sejam removidas da maior fila de execução até que a diferença de tamanho entre as duas filas de execução se reduza à metade. Para reduzir a sobrecarga, o balanceamento de carga não é executado a menos que a fila de execução cuja carga seja mais pesada contenha 25% mais tarefas do que a fila de execução do processador que está realizando o balanceamento de carga. (Deitel, 2005)

Quando o escalonador seleciona tarefas para balanceamento, tenta escolher aquelas cujos desempenhos serão menos afetados por passarem de um processador para outro. Em geral, a tarefa menos recentemente ativa de um processador é a que terá mais probabilidade de estar fria em relação ao *cache* do processador, uma tarefa fria em relação ao *cache* não contém muitos (ou nenhum) dos dados da tarefa no *cache* do seu processador, ao passo que uma tarefa quente em relação ao *cache* contém a maioria ou todos os dados da tarefa no *cache* do processador. Portanto, o escalonador escolhe migrar tarefas que provavelmente são mais frias em relação ao *cache*.

ESCALONAMENTO DE TEMPO REAL

O escalonador suporta escalonamento de tempo real tentando minimizar o tempo durante o qual uma tarefa de tempo real espera para ser selecionada para um processador. Diferentemente de uma tarefa normal, que é eventualmente colocada na lista expirada para impedir que tarefas de baixa prioridade sejam indefinidamente adiadas, uma tarefa de tempo real é sempre colocada na lista ativa depois que seu *quantum* expirar. Além disso, tarefas de tempo real sempre executam com prioridades mais altas do que tarefas normais. Porque o escalonador sempre seleciona uma tarefa da fila de maior prioridade da lista ativa, tarefas normais não podem provocar a preempção de tarefas de tempo real.

O escalonador obedece à especificação POSIX para processos de tempo real, permitindo que tarefas de tempo real sejam escalonadas por meio do algoritmo de escalonamento de alternância circular ou FIFO. Se uma tarefa especificar escalonamento por alternância circular e seu intervalo de tempo tiver expirado, receberá um novo intervalo de tempo e entrará no final da fila de seu arranjo de prioridades da lista ativa. Se a tarefa especificar escalonamento FIFO, não receberá um intervalo de tempo e, portanto, executará em um processador até sair, adormecer, bloquear ou ser interrompida.

Processos de tempo real podem adiar indefinidamente outros processos caso sejam codificados inadequadamente, resultando em tempos de resposta ruins. Para evitar

utilização imprópria, acidental ou mal-intencionada de tarefas de tempo real, apenas usuários com privilégios de *root* (administrador) podem criá-las.

CONSIDERAÇÕES FINAIS

A política de escalonamento em Linux procura satisfazer os objetivos conflitantes que surgem em todos os sistemas operacionais, ou seja, oferece um tempo de resposta rápido, procura executar o maior número de tarefas no menor espaço de tempo e concilia processos de alta prioridade com os de baixa prioridade. Por tratar processos e *threads* como tarefas, internamente ambos são representados por uma única tarefa, assim, *threads* simplificam o código do núcleo e reduzem carga requisitando somente uma cópia das estruturas de dados de gerenciamento de tarefas. Por usar o escalonamento por preempção, o Linux garante uma boa parte de tempo para cada processo, garantido assim uma distribuição de tempo uniforme.
Uma meta interessante do escalonador é impedir adiamento, estabelecendo um período de tempo denominado época durante o qual cada tarefa da fila de execução executará no mínimo uma vez, impedindo que ocorra o *starvation*. O reescalonamento permite que tarefas *I/O bound* e tarefas interativas executem mais do que uma vez por época, evitando uma espera de tempo. Para evitar a situação de ociosidade em multiprocessadores, se o escalonador detectar que um processador esteja ocioso, executará balanceamento de carga para migrar tarefas de um processador para outro de modo que melhore a utilização de recursos. Se o sistema tiver apenas um processador, as rotinas de balanceamento são eliminadas do núcleo quando este é compilado. Em escalonamento de tempo real, tarefas de tempo real sempre executam com prioridades mais altas do que tarefas normais. Porque o escalonador sempre seleciona uma tarefa da fila de maior prioridade da lista ativa, tarefas normais não podem provocar a preempção de tarefas de tempo real. Finalizando, a política de escalonamento de processos em Linux, reúne todos os métodos de escalonamento, levando em consideração, o escalonamento por múltiplas filas com realimentação, onde a prioridade dos processos é reavaliada, beneficiando todos os processos.

REFERÊNCIAS

Deitel, H.M. **Sistemas Operacionais**, 3ª edição, Pearson Education do Brasil, 2005.

Oliveira, R.S., Carissimi, A.S., Toscani, S.S., **Sistemas Operacionais**, 3ª edição, Sagra Luzzato, 2004

Machado, F.B., Maia, L.P., **Arquitetura de Sistemas Operacionais**, 2ª edição, LTC,

Tanenbaum, A.S, **Sistemas Operacionais Modernos**, 2ª edição, Pearson Education do Brasil, 2003

Open Source Development Labs, Inc., ***Linux Process Scheduler Improvements in Version 2.6.0***, Disponível em http://developer.osdl.org/craiger/hackbench/.

Rick Lindsley, ***What's New in the 2.6 Scheduler***, Disponível em http://www.linuxjournal.com/article.e.php?sid=7178

Real, L.C.V., Damasio, F., **Escalonamento interativo no kernel Linux**, Universidade do Vale do Rio dos Sinos Centro de Ciências Exatas e Tecnológicas

Toledo, Marcelo. **Gerenciamento de Processos no Linux**. 2005

Fávero, André Luiz. **Suporte a processadores simétricos (SMP) em kernel Linux**. UFGRS.

Starke, Márcio. Maziero, Márcio. Jamhour, Edgard. **Controle Dinâmico de Recursos em Sistemas Operacionais**. PUC-PR

T. Aivazian, **Linux kernel 2.4 internals**, 2001, disponível em: www.tldp.org/ldp/lki/lki.htlm

D. Rusling, **The Linux kernel**, 1999, disponível em: www.tldp.org/ldp/tlk/tlk.html

A. SchlapBach, **Linux process scheduling**, 2000, disponível em: www.unibe.ch/sdutenten/schlbch/linuxscheduling/linuxscheduling.htm

S.Walton, **Linux threads frequently asked questions**, 1997, disponível em: www.tldp.org.com/faq/threads-faq

D. McCracken, **POSIX threads amd the Linux kernel**, proceedings of the Ottawa Linux Symposium, 2002

R. Arcomano, **Kernel Analysis Howto**, 2002, disponível em: www.tldp.org/howto/kernelanalysis-howto.html

S. Walton, **Linux threads frequently asked questions**, 2002, disponível em: www.linas.org/linux/threads-faq.html

U. Drepper e I. Molnar, **The native POSIX threads library for Linux**, 2003, disponível em: http://people.redhat.com/drepper/nptl-design.pdf

I. Molnar, **Announcenment to Linux mailling list**, 2003, disponível em: http://lwn.net/2002/0110/a/scheduler.php3

Cross-referecing Linux, disponível em: http://lxr.linux.no/source/kernel/sched.c?v=2.5.56

Linux kernel source code, versão 2.6.0-test2, disponível em: http:// .linux.no/source/kernel/sched.c?v=2.6.0-test2

Linux kernel source code, versão 2.5.75, disponível em: http://lxr.linux.no/source/kernel/sched.c?v=2.5.75

Recebido: 10/08/2008
Aceito: 14/10/2008