Os destaques do kernel 3.0

Por dentro do kernel

Como são as entranhas do kernel Linux 3.0? Analisamos algumas das melhorias desde a versão 2.6.0 para que você aproveite os principais recursos e entenda porque a arquitetura do Linux torna um sistema operacional em estado da arte.

por Eva-Katharina Kunst e Jürgen Quade

Segundo Linus Torvalds, a versão 3.0 do Linux se refere "apenas à renumeração". Em seu email para a comunidade de desenvolvedores de 29 de maio, que acompanhou o release candidate do novo kernel [1], Linus acabou com a especulação de que haveria novidades e capacidades surpreendentes. Ele descreveu a versão como não tendo "nenhuma mudança de ABI (*application binary interface*), nenhuma mudança de API (*application programming interface*), nenhum mágico novo recurso, apenas progresso rotineiro e firme ... mudanças de drivers (o grosso realmente são atualizações de drivers)".

Se você comparar o Linux 3.0 com o kernel mais recente da série 2.6, o 2.6.39, Linus certamente está certo sobre a abrangência limitada das mudanças. Mas se você comparar com o kernel 2.6.0, de oito anos atrás, verá que a versão 2.6 introduziu melhorias revolucionárias. Decidimos que a rara ocasião de uma grande nova versão Linux seria uma ótima oportunidade para um passeio pelo kernel e uma boa olhada em algumas das mudanças recentes e mais importantes.

Dançando em volta do monolito

Como nos primeiros sistemas Unix, o Linux tem uma estrutura monolítica. Todos os serviços centrais residem em um componente principal, conhecido como *kernel*. Embora os defensores da arquitetura alternativa de *microkernel* argumentem que um kernel monolítico aumenta o risco de instabilidade, os benefícios de desempenho de um design monolítico ultrapassam de longe as desvantagens – pelo menos em configurações de produção, já que um único kernel mantém os componentes juntos e elimina a espera causada por mudanças de contexto (de qualquer modo, apesar dos alertas em relação a esse design monolítico, o kernel Linux sempre gozou da reputação de ótima estabilidade).

A **figura 1** é uma visão geral básica da estrutura de sistema do Linux. A interface para chamadas de sistema define o tipo e o número de serviços que um aplicativo pode requisitar do sistema operacional: iniciar ou terminar tarefas, armazenar ou ler dados, ou usar o protocolo TCP/IP para enviar dados a outros computadores. O kernel 2.6.0 oferecia 274 serviços no Linux 32-bit; o 3.0 tem 347.

Os serviços em si são fornecidos pelos blocos subjacentes de gerenciamento de entrada e saída, de tarefas e de memória, que usam drivers de dispositivos para acessar o hardware. Uma subdivisão mais granular revela os subsistemas do kernel responsáveis por tarefas como multimídia, rede ou acesso a dispositivos USB ou PCI.

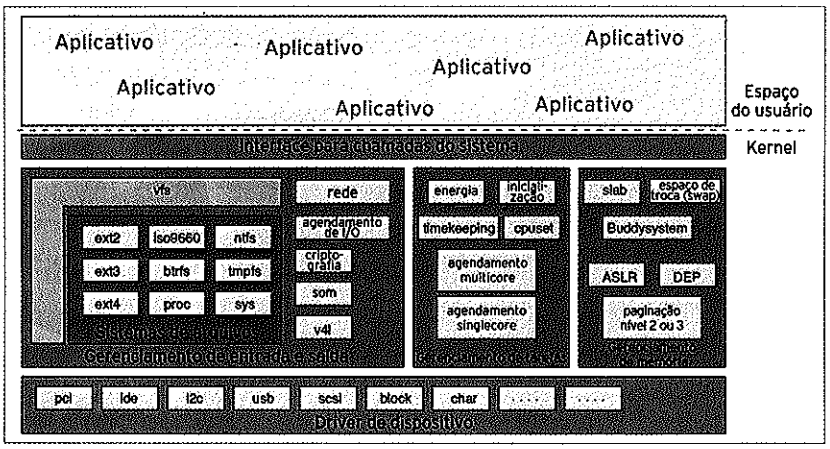

Figura 1 O kernel 3.0 pode ter arquitetura monolítica, mas por dentro goza de uma estrutura totalmente modular.

Nas seções a seguir, vamos conhecer mais de perto os serviços oferecidos pelos blocos principais da **figura 1**:

- Gerenciamento de tarefas
- Gerenciamento de memória
- Gerenciamento de entrada e saída

Durante o caminho, você aprenderá sobre alguns avanços importantes do kernel 2.6, incluindo melhorias no processamento em tempo real e o recurso *tickless*.

Gerenciamento de tarefas

Dentro do kernel, o gerenciamento de tarefas tem papel central: ele é o responsável pela multitarefa, o processamento simultâneo de vários programas. Compatibilidade com máquinas de múltiplos núcleos tem sido um dos pontos fortes do Linux por muitos anos. O agendamento – isto é, decidir qual tarefa é executada por quanto tempo em qual núcleo – é um processo de duas etapas: o agendador de múltiplos núcleos agrupa processadores individuais ou os núcleos de uma máquina multicore. Dentro dos grupos, as tarefas são designadas para as CPUs. Na segunda etapa, que é o agendamento de núcleo único, cada processador escolhe as tarefas que pode executar a partir das tarefas designadas para ele.

Antes do agendador de múltiplos núcleos se tornar ativo, o kernel mapeia o hardware existente na fase de boot e os designa hierarquicamente a domínios de agendamento, devido à arquitetura multicore (*hyperthreading*, SMP e arquitetura de memória não uniforme). Domínios de agendamento contém grupos de agendamento que, por sua vez, contém núcleos de CPU ou outros domínios de agendamento. Uma lista de tarefas é designada para cada grupo ou núcleo.

O agendador consegue uma distribuição balanceada da carga através da migração de processos, ou seja, movendo tarefas para outro núcleo ou outro grupo dentro do domínio. O agendamento de múltiplos núcleos está sempre ativo: quando uma tarefa termina, uma nova tarefa inicia, quando uma tarefa dorme ou no caso de um desnível entre as cargas designadas para os núcleos., o Linux sempre calcula se uma migração de processos terá benefícios e o administrador de sistema pode influenciar esse comportamento usando uma opção de linha de comando.

No kernel 2.6.23, foi introduzida uma biblioteca extensível para o agendamento de núcleo único, com o objetivo de facilitar a implementação de algoritmos de agendamento. A biblioteca introduziu classes de agendamento, que dependem do algoritmo implementado para escolher a próxima tarefa a ser processada, a partir da fila de tarefas designadas. Se uma classe falhar ao retornar uma tarefa após requisição, o agendador apenas pede a próxima classe.

O kernel padrão implementa três classes: a `rt_sched_class` é responsável por processos em tempo real. O algoritmo dessa classe implementa o agendamento de prioridade. Se múltiplas tarefas têm a mesma prioridade, aplicam-se as lógicas FCFS (*First Come First Served*) ou *Round-Robin*. No total, o Linux oferece 99 níveis de prioridade nessa classe.

Tarefas que iniciam de modo normal são designadas para a `fair_sched_class`, que é implementada pelo *Completely Fair Scheduler* (CFS). Esse algoritmo deixa de ordenar os processos executáveis em listas – como nas primeiras versões 2.6 – usando no lugar disso árvores rubro-negras. Esse design permite uma distribuição adequada de recursos de processamento com overhead mínimo, não dependendo mais de heurística, mas de matemática pura. Níveis *nice* [2], que os primeiros

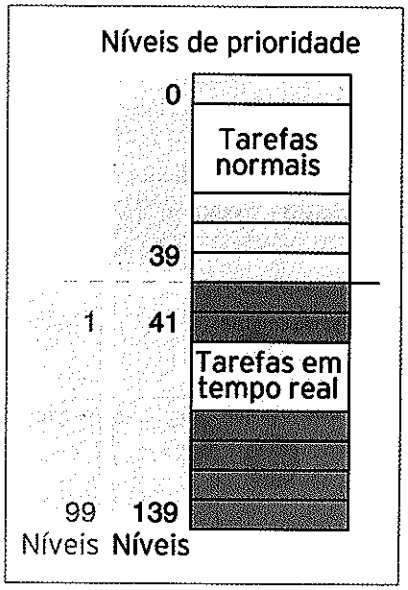

Figura 2 De um ponto de vista lógico, o Linux tem 140 níveis de prioridade, que são subdivididos em espaços para tarefas normais ou em tempo real.

Quadro 1: Modelo de interrupção

O Linux distingue entre quatro níveis para processar código (**figura 4**). Usuários estão restritos ao nível do usuário e aos aplicativos que executam aí. Tarefas que executam no espaço do usuário podem entrar em estado de dormência. Em máquinas com múltiplos núcleos, as funções podem executar múltiplas vezes em paralelo.

Chamadas de sistema e threads do kernel – que têm propriedades similares – ficam no espaço do kernel, podem ter preempção (interrupções bloqueadas) em execução quase paralela em máquinas de um núcleo e execução paralela genuína em máquinas multicore, além da capaciade de dormência.

Colocar em repouso é algo realmente proibido no nível do IRQ de software. É aí que timers e tasklets (pequenas sequências de código) que permitem interrupções residem; essa área era antigamente conhecida como "a metade de baixo".

O nível mais inferior é ocupado pelas rotinas de interrupção de serviços (ISR, na sigla em inglês), que podem interromper qualquer sequência de código de níveis superiores. Mesmo assim, elas não são tipicamente passíveis de preempção, a menos que isso seja explicitamente permitido.

sistemas Unix implementaram para dar suporte à prioridade de tarefas entre -20 e 19, ainda são visíveis no espaço do usuário (com o comando `nice`). O comando `ps -ce` mostra os níveis nice como prioridades Linux entre 0 e 39.

As prioridades de tempo real (1 a 99) retornam aqui como prioridades de 41 a 139; então você sempre precisa subtrair 40 ao converter as prioridades Linux em prioridades de tempo real POSIX (**figura 2**).

Finalmente, a terceira classe é a `iddle_sched_class`, que só fica ativa se não houver nenhuma outra tarefa executável.

A preempção pelo agendador no kernel Linux também tem a ver com funções que são processadas no kernel ou no contexto de processo (**quadro 1**), ou seja, chamadas de sistema ou threads do kernel. A abordagem clássica é o Linux concluir as funções que o kernel iniciou antes de começar a lidar com outras funções. Isso também se aplica no caso de uma função muito importante precisar ser processada – por exemplo, como indicado por uma interrupção.

Esse cenário é diferente com o kernel moderno: funções do kernel com prioridades mais altas interrompem funções com prioridades baixas. Esse design reduz consideravelmente os tempos de latência.

As interrupções em thread (**figura 3**) são suplementos úteis à preempção de kernel. Se um kernel 3.0 for iniciado com a opção `threadirqs`, o sistema operacional processa rotinas de interrupção de serviço (ISR) no contexto do kernel, ou seja, como threads de kernel. Em outras palavras, o administrador de sistema pode não apenas influenciar a prioridade de interrupções, mas pode fazer isso também em relação a outras tarefas.

Para permitir que o agendador se torne ativo, o Linux precisa de interrupções. Por esse motivo, versões anteriores do Linux incluíam um timer que gerava periodicamente uma interrupção (por exemplo, a cada 10 milissegundos). No Linux, o número de interrupções é contado com a variável `jiffies`, sendo assim equivalente a frequência (*heartbeat*) do kernel. Torvalds reduziu o intervalo para 4 milissegundos na versão 2.6, aumentando assim consideravelmente o comportamento de interação. No entanto, esse intervalo menor acaba afetando a eficiência, já que o kernel precisa estar ativo com mais frequência.

A opção tickless do kernel surgiu primeiro na versão 2.6.21. O kernel 3.0 agora é tickless, ou seja, interrupções não são mais criadas periodicamente, apenas quando necessário. A lista de vantagens impressiona: o design tickless reduz a carga no sistema, aumenta a eficiência e – principalmente em dispositivos móveis – fornece períodos maiores de inatividade, economizando energia.

Não é preciso dizer que há um preço por isso: o kernel precisa calcular o próximo ponto no tempo para cada interrupção e, depois, reprogramar o componente de tempo, além de atualizar o tempo interno básico. Os desenvolvedores do Linux puderam reconstruir a contagem do tempo com precisão de nanossegundos, com base nesse sistema tickless.

Como mostra a **figura 5**, tarefas sempre terminam a si mesmas. Contudo, essa chamada suicida pode ser emitida por um sinal de outra tarefa (usando a chamada de sistema `kill`). Antes de todos os *traces* da tarefa serem removidos, ela muda para um estado zumbi. Nesse estado, o kernel já liberou a área de memória correspondente

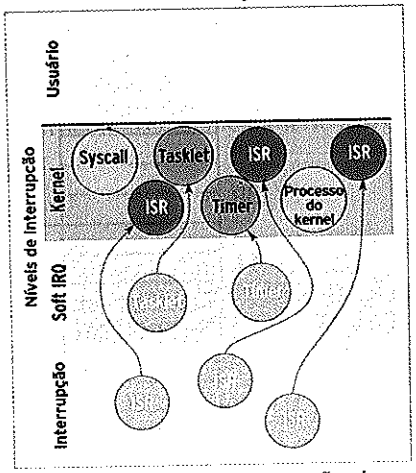

Figura 3 Rotinas de interrupção de serviços (ISRs) e IRQs de software tornam as threads interrompíveis. Um processo em tempo real executando no espaço do usuário pode desalojá-los.

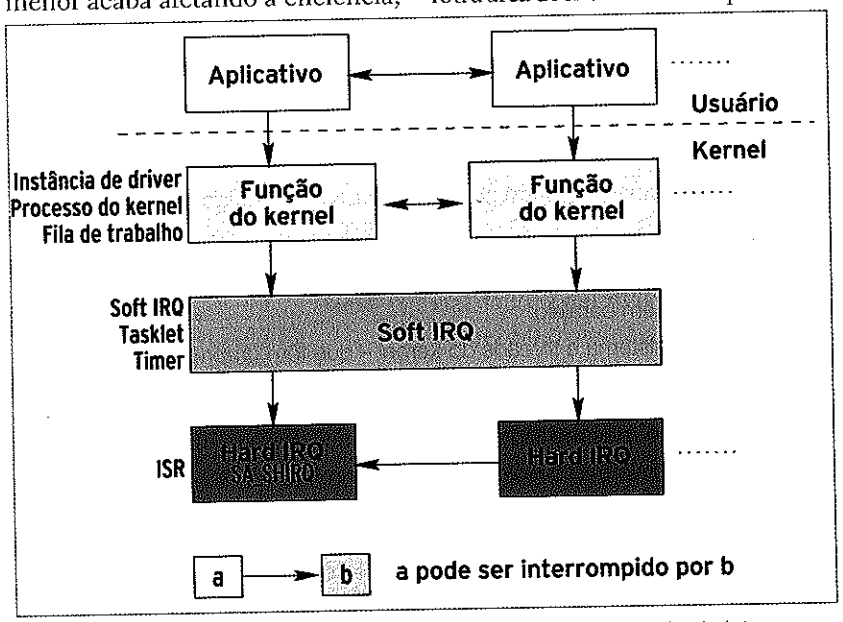

Figura 4 Programadores do kernel precisam conhecer o modelo de interrupção para identificar áreas críticas e protegê-las corretamente.

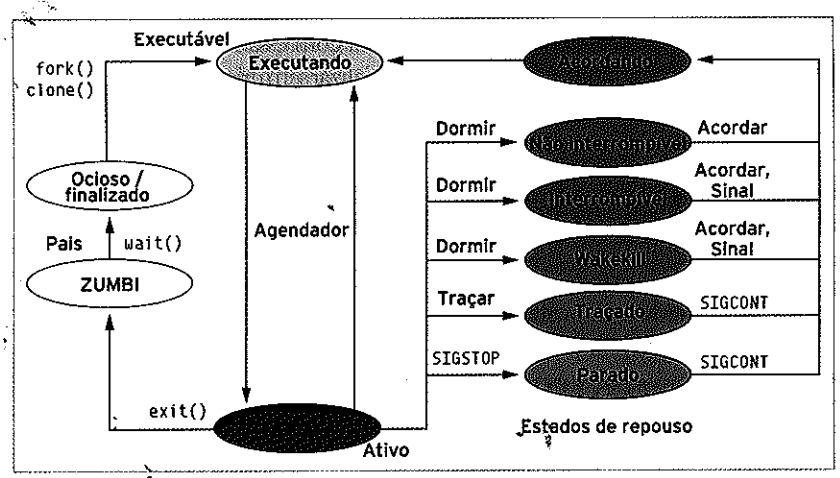

Figura 5 Aplicativos e threads do kernel passam por diversos estados.

(segmentos de código, dados e stack); contudo, o bloco de controle de tarefas (a estrutura de dados que representa a tarefa no kernel) ainda existe. Antes de ser apagada, a tarefa-mãe precisa pegar o código de saída (o valor de retorno de `main`). Se a própria tarefa-mãe tiver terminado, o processo init (a tarefa com o PID de 1) cuida disso.

Gerenciamento de memória

O segundo maior componente do kernel é o gerenciamento de memória. Uma importante responsabilidade do gerenciamento de memória é implementar o endereçamento. O kernel garante que todo aplicativo possa acessar a memória principal, que começa no endereço 0 (zero) e contém 3 ou 4 GB de tamanho – ou até mais, dependendo da configuração. Um sistema 32-bits (plataforma PC) tipicamente irá implementar o gerenciamento de memória com paginação de dois níveis; uma máquina 64-bits terá paginação de três níveis. Por questões de segurança, o Linux torna aleatórias as posições de endereço dos segmentos individuais de um aplicativo (*heap*, *stack* e bibliotecas compartilhadas). O fato de que o Linux tem um *Address Space Layout Randomization* (ASLR) mais pronunciado que outros sistemas é um importante recurso de segurança.

A proteção, outra tarefa do gerenciamento de memória, é similar. Aplicativos não tem permissão para acessar áreas da memória usadas por outros aplicativos ou pelo espaço do kernel para, digamos, pegar as senhas armazenadas na memória principal. Para implementar a proteção de memória, o Linux é compatível com os mecanismos definidos pelo hardware. Essa abordagem também se aplica ao evitar a execução de código na pilha (*Data Execution Prevention*, DEP) – outra boa propriedade que protege contra ataques maliciosos. Infelizmente, a maioria das distribuições não se dá o trabalho de habilitar esse recurso em suas versões 32-bits.

O Linux depende de objetos tipificados e se beneficia de reciclagem graças ao alocador slab (**figura 6**). O kernel fornece diversos objetos pré-inicializados. Essa abordagem tem benefícios para o desempenho: a inicialização simultânea de múltiplos objetos idênticos otimiza o uso do cache do processador. Ela melhora o uso da memória existente e acaba com a necessidade de código extra, já que as operações que alocam memória, pré-inicializam objetos, desfazem a inicialização e liberam memória não mais necessária. Administradores podem visualizar dados desse tipo com o comando `cat /proc/slabinfo`.

Gerenciamento de entrada e saída

O gerenciamento de entrada e saída (*input/output*, ou I/O) é responsável pelo acesso a arquivos e dispositivos periféricos. O ponto de distribuição central para isso é o *Virtual Filesystem Switch* (VFS), que distribui o acesso e repassa as requisições de disco rígido para o agendador I/O. Ele é responsável por organizar as requisições de modo inteligente para que a cabeça de leitura de um disco rígido faça o menor número possível de movimentos para trás. Como os movimentos de cabeça são totalmente irrelevantes para SSD (memória flash), o Linux irá passar dados diretamente para o disco se for configurado assim.

O VFS também implementa o modelo hierárquico que o Linux usa para gravar ou ler dados de um disco. Nesse modelo, cada arquivo é representado

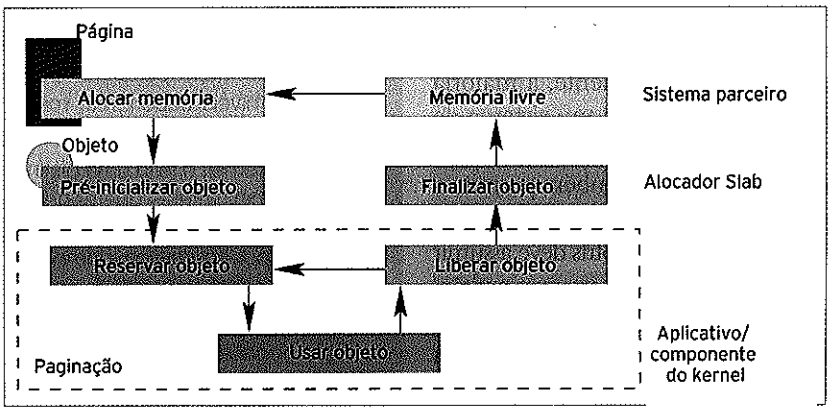

Figura 6 O ciclo de vida de objetos do kernel: ao usar bastante a reciclagem de objetos, o kernel evita a necessidade de reservar e inicializar memória o tempo todo.

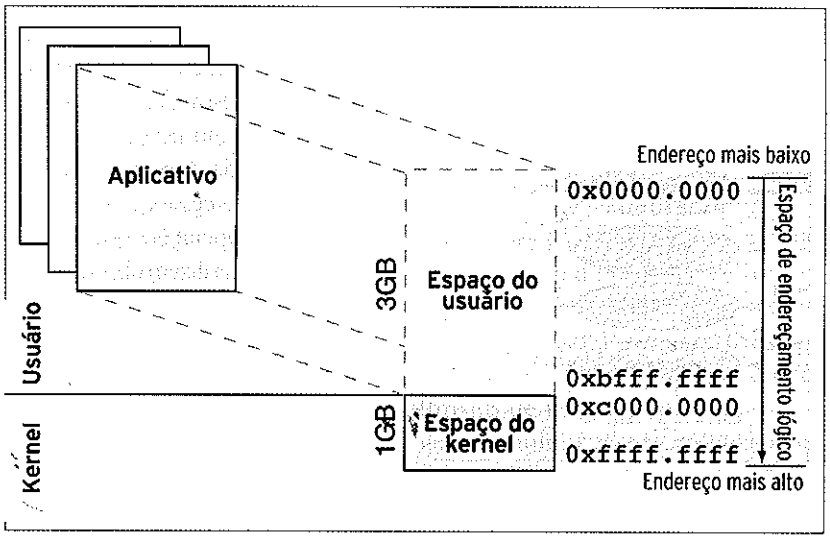

Figura 7 Como 25% todos os diretórios de paginação têm o mesmo conteúdo, não há necessidade de alternar áreas de memória na mudança para o espaço do kernel.

por um *inode*. O inode, que tem um número único, armazena os meta-dados: tamanho do arquivo, data de criação, último acesso, privilégios de acesso, propriedade e tudo mais. Contudo, esses meta-dados não têm um nome único. Incidentalmente, você pode listar números inode digitando `ls -i`.

Um objeto chamado de *dentry* liga o nome do arquivo ao inode. Para poder usar nomes para acessar os arquivos, você precisa primeiro pesquisar as dentries. O Linux acelera esse longo processo usando memória temporária conhecida como *dcache*.

Alguns métodos estão ligados ao objeto inode. Para resumir, eles cuidam do mapeamento de disco e das estruturas de dados que residem ali – em outras palavras, o sistema de arquivos. Atualmente, a maioria das distribuições usa um sistema de arquivos ext4, que não só tem bom desempenho mas também permite o armazenamento de arquivos grandes.

Com o tempo, o Btrfs deve assumir o papel de sistema de arquivos padrão. Quando isso acontecer, verificações no sistema de arquivos serão possíveis com o sistema montado e em uso. A interface do sistema de arquivos universal permite ao Linux acessar diversos sistemas de arquivos, como FAT, NTFS ou ISO9660.

Por cima disso tudo, dados podem ser armazenados em sistemas de arquivos virtuais: do ponto de vista do aplicativo há arquivos que só existem na memória RAM e não no disco. E os dados não são criados até que o acesso ocorra. Administradores de sistema tem familiaridade com três sistemas virtuais: o sistema de arquivos *process* contém dados sobre processos de computação; o sistema de arquivos *sys* tem os detalhes da estrutura de hardware, incluindo os respectivos drivers; e finalmente, o sistema de arquivos *temp* permite que dados sejam armazenados na memória principal sem envolver o trabalho de um sistema de arquivos clássico. O Linux conta com essa habilidade para poder inicializar sem drivers usando o *initramfs*. Além disso, o VFS permite camadas de sistemas de arquivos para fornecer uma solução simples de criptografia.

Os dispositivos de drivers estão intimamente relacionados ao VFS, sendo o "filé mignon" do código-fonte do Linux. Historicamente, um driver é identificado com um número de 8-bits. Os 256 drivers permitidos assim definitivamente não são suficientes para um sistema operacional moderno. O Linux então trabalha hoje com números de dispositivos de 32-bits, que suficientemente permitem a codificação de 4.000 drivers e uma base de 4 bilhões de dispositivos endereçáveis.

Melhorias em tempo real

O grande salto do kernel entre as versões 2.6 e 3.0 está ligado ao seu comportamento de tempo real. O desenvolvedor alemão Thomas Gleixner foi um grande contribuidor para os avanços nessa área. Ele empacotou seus patches de tempo real em "cavalos de Tróia digeríveis" e assim deu aos arquitetos algumas opções atrativas para melhorar os registros do Linux como um sistema em tempo real: timers de alta resolução (*hrtimers*) permitem lidar com tempo de modo extremamente preciso; *mutexes* de tempo real permitem a herança de prioridades

Quadro 2: Interrupções

O modelo de interrupção mostra não apenas que níveis inferiores podem interromper níveis superiores, mas também o contexto em que um nível é processado. ISRs e IRQs de software executam no contexto de interrupção.

No nível do kernel, é feita uma distinção entre o contexto de processo e o contexto do kernel. Tarefas que pertençam a um aplicativo no espaço do usuário executam todas no contexto do processo. Elas podem ser chamadas de sistema feitas por aplicativos, como `open`, `close`, `read` e `write`. Essas sequências de código podem trocar dados entre o kernel e o aplicativo correspondente.

Isso deixa as threads do kernel para o contexto do kernel. Cada contexto tem uma definição no cabeçalho do kernel (`GFP_ATOMIC` para o contexto de interrupção, `GRP_KERNEL` para o contexto do kernel e `GFP_USER` para o contexto de processo). Funções genéricas como `kmalloc()` (`malloc()` no kernel) podem ser invocadas a partir de diferentes contextos. Programadores do kernel precisam informar o contexto atual com uma definição, para garantir que a função saiba se pode ou não dormir.

e novas CPUs designam processos de cálculo exclusivamente a threads específicas de sistemas multicore. E isso se soma às técnicas já mencionadas neste artigo, incluindo interrupções como processos, preempção do kernel e o recurso tickless.

O Big Kernel Lock (BKL), notório entre os programadores do kernel, que simplesmente bloqueava as interrupções em todos os processadores, também desapareceu do kernel a tempo para o lançamento da versão 3.0.

Conclusão

O kernel Linux 3.0 é uma excitante peça de software para produção – e não apenas porque é orientado a objetos sem realmente usar uma linguagem orientada a objetos. O kernel é incomumente escalonável e portável, e seu modelo de desenvolvimento de código aberto combina as ideias e habilidades de programação de desenvolvedores de todo o mundo em um sistema confiável, flexível e altamente eficiente. ■

Quadro 3: Estados de tarefas

Tarefas podem ter diferentes estados (**figura 5**). Uma nova tarefa sempre é criada pela chamada de sistema `clone()` como uma cópia exata de seu processo-mãe. A nova tarefa tem o estado `TASK_RUNNING`. O agendador escolhe a próxima tarefa a ser processada a partir das tarefas disponíveis. Uma mudança de contexto ativa a tarefa, que então muda o estado para ativo. Uma tarefa irá entrar em repouso, ou dormir, por vários motivos: pode estar aguardando dados, aguardando recursos ou simplesmente esperando algum tempo passar.

O Linux distingue entre dois tipos de repouso. O modo mais simples é a tarefa dormir até o fim da condição de repouso, quando algo no kernel dispara uma chamada `wake_up`. Nesse estado, a tarefa não vai despertar com nada mais além disso. Já no estado "matável", a tarefa não fica dormindo tão profundamente. Pelo menos, você pode despertá-la emitindo um SIG 9 (`SIGKILL`). "Interrompível" é a forma mais leve de repouso, em que a tarefa pode despertar com qualquer sinal.

Quadro 4: Paginação

O gerenciamento de memória converte os endereços lógicos usados pelos aplicativos em endereços físicos. O principal método usado para isso é conhecido como paginação. A paginação divide a memória principal em páginas do mesmo tamanho. Uma página tipicamente tem 4 KB. Em outras palavras, 4 GB de RAM são divididos em um milhão de páginas de memória. Para endereçar um milhão de páginas, você precisa de 20 bits, e para encontrar uma célula de memória dentro de uma página você precisa de mais 12 bits.

Para mapear os 20 primeiros bits de um endereço lógico para os 20 primeiros bits de um endereço de página, cada tarefa tem um diretório de página; na plataforma PC, o diretório de página é uma área de memória de 4 KB (1.024 entradas de 4 bytes cada). Além disso, pode haver até 1.024 tabelas de páginas, cada uma com 1.024 entradas e 4 bytes cada.

O endereço de 32-bit usado pelo aplicativo é dividido em três áreas: os primeiros 10 bits (bits 22 a 31) selecionam uma das 1.024 células de memória no diretório de página (2^10 = 1.024). O valor ali armazenado é endereçado por uma das 1.024 tabelas de páginas. O segundo grupo de 10 bits (bits 12 a 21) do endereço lógico representam o índice para a tabela de página e seleciona a célula de memória retornada pelo endereço de 20-bits da página física. Os últimos 12 bits então endereçam a memória dentro da página selecionada.

Como essa abordagem em dois estágios consome tempo, processadores armazenam as conversões já realizadas no *Translation Lookaside Buffer* (TLB). No entanto, esse buffer precisa ser apagado a qualquer mudança de contexto, já que cada tarefa tem sua própria implementação.

Quadro 5: Patch 4G/4G

Em um sistema 32-bit, os registradores de endereço têm uma largura de 32 bits, podendo assim acessar 4 GB de RAM virtual. Aplicativos que endereçam todo esse espaço e tentam acessar áreas acima dos 3 GB estão fadados ao fracasso, provocando um erro de segmentação.

Os desenvolvedores reservaram para o kernel os gigabytes mais altos do espaço de endereçamento do aplicativo. Ou seja, os 25% de cima do diretório de página de cada tarefa são iguais para todas as tarefas, enquanto o restante é específico de cada tarefa (**figura 7**). Isso significa que o kernel pode usar o diretório de página de qualquer tarefa ativa para acessar as áreas de memória que ele referencia ali: o espaço do kernel.

Sem esse truque, o diretório de página, ou seja, o ponto de entrada do gerenciamento de memória, teria que ser substituído para cada chamada de sistema e para cada interrupção. Isso também incluiria a necessidade de zerar os *Translation Lookaside Buffers* (TLB) e o cache do gerenciamento de memória, o que consumiria muito tempo.

Se uma tarefa que usa muita memória realmente precisar de 4 GB de memória virtual, você pode habilitar o patch 4G/4G, que designa um diretório de página separado para o kernel. Mas nesse caso, em vez disso talvez você prefira mudar para um sistema de 64 bits.

Mais informações

[1] Linus Torvalds, "Linux 3.0-rc1": `http://thread.gmane.org/gmane.linux.kernel/1147415`

[2] Níveis nice: `http://en.wikipedia.org/wiki/Nice_%28Unix%29`

Gostou do artigo?

Queremos ouvir sua opinião. Fale conosco em cartas@linuxmagazine.com.br

Este artigo no nosso site: http://lnm.com.br/article/5900