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

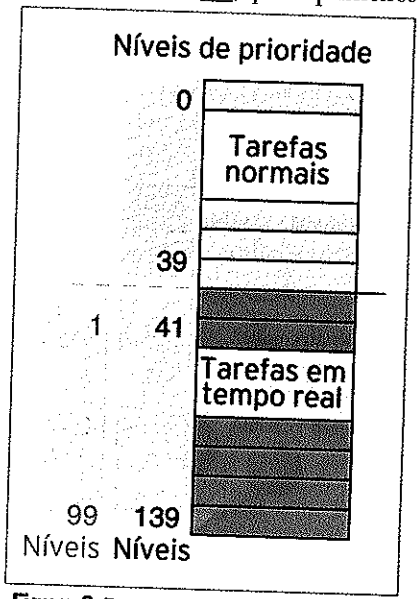

Figura 2 De um ponto de vista lógico, o Linux tem 140 níveis de prioridade, que são subdivididos em espaços para tarefas normais ou em tempo real.

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

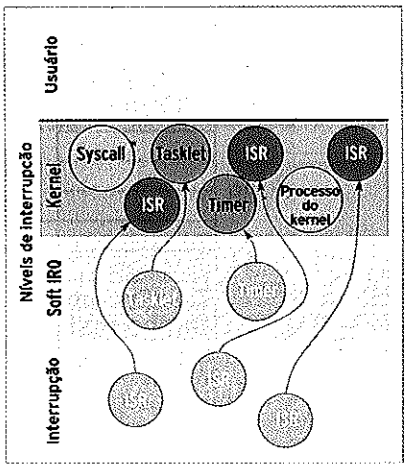

Figura 3 Rotinas de interrupção de serviços (ISRs) e IRQs de software tornam as threads interrompíveis. Um processo em tempo real executando no espaço do usuário pode desalojá-los.

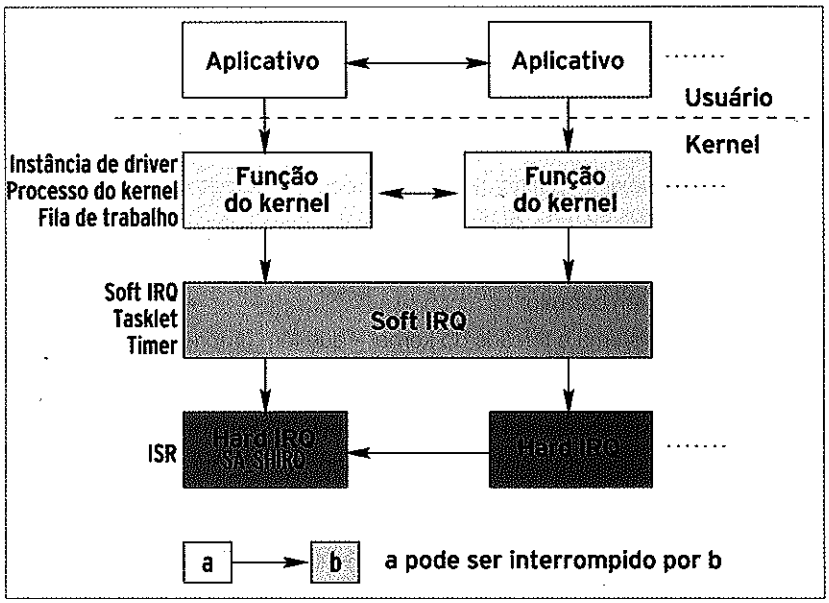

Figura 4 Programadores do kernel precisam conhecer o modelo de interrupção para identificar áreas críticas e protegê-las corretamente.

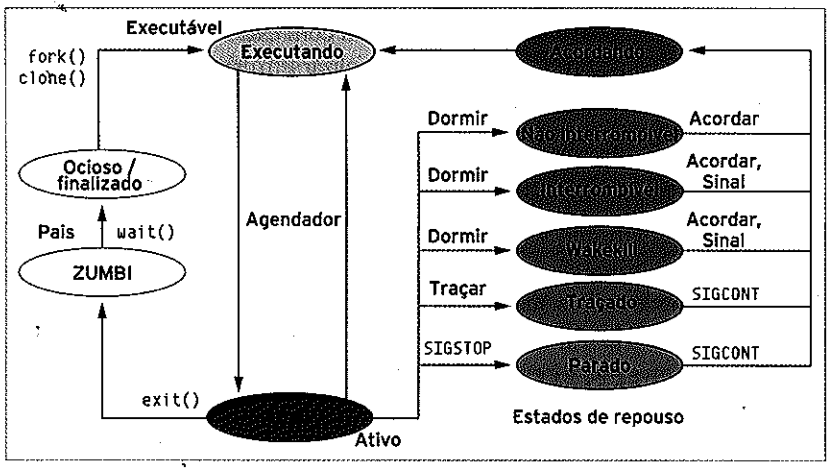

Figura 5 Aplicativos e threads do kernel passam por diversos estados.

(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

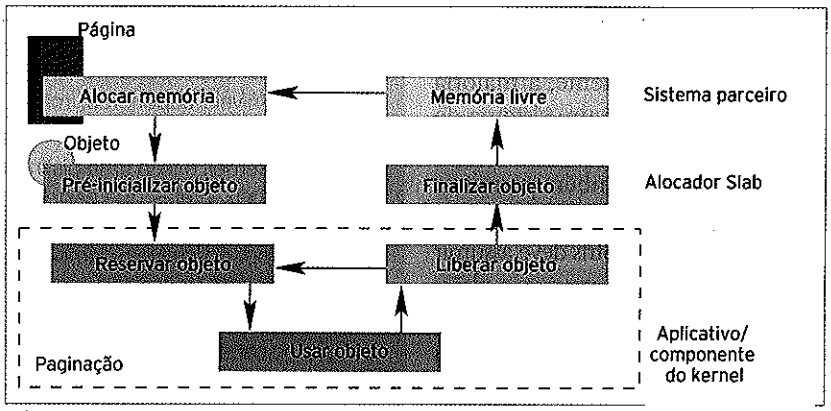

Figura 6 O ciclo de vida de objetos do kernel: ao usar bastante a reciclagem de objetos, o kernel evita a necessidade de reservar e inicializar memória o tempo todo.