Um estudo do VFS do Kernel Linux

Daniel Higa, Felipe Soares, Brivaldo Alves Silva Jr

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79.070-900 – Campo Grande – MS – Brazil

{daniel.higa, felipe.soares,brivaldo.junior}@ufms.br

Abstract. *This work aimed to deepen the theoretical knowledge of the Linux operating system kernel, with a special focus on the Virtual Filesystem (VFS), a core component of the kernel. To this end, a detailed study of the Linux kernel version 6.13.2 was conducted, exploring its main data structures and internal mechanisms that support the operation of the VFS, such as inodes, dentries, files, and superblocks. Fundamental concepts for kernel module development, process management and scheduling, as well as the functioning of system calls and the Unix-like model of Linux were also addressed. The research involved analyzing memory allocation mechanisms, such as the Slab Allocator, and studying zombie processes, aiming to understand common practices for module development with a view to subsequent integration with an academic filesystem.*

Resumo. *Este trabalho teve como objetivo aprofundar o conhecimento teórico sobre o núcleo do sistema operacional Linux, com foco especial no Virtual Filesystem (VFS), componente essencial do kernel. Para isso, foi realizado um estudo detalhado do kernel Linux versão 6.13.2, explorando suas principais estruturas de dados e mecanismos internos que suportam o funcionamento do VFS, como inodes, dentries, arquivos e superblocks. Também foram abordados conceitos fundamentais para desenvolvimento de módulos de kernel, gerenciamento e escalonamento de processos, além do funcionamento das system calls e do modelo Unix-like do Linux. A pesquisa envolveu a análise de mecanismos de alocação de memória, como o Slab Allocator, e o estudo dos processos zumbis, visando compreender práticas comuns para o desenvolvimento de módulos para subsequente integração com um sistemas de arquivos acadêmico.*

1. Introdução

A atividade teve como objetivo promover o desenvolvimento e o domínio teórico sobre o núcleo de sistemas operacionais baseados em Linux, com ênfase especial na compreensão do Virtual Filesystem (VFS) — um dos componentes centrais do kernel Linux. Para isso, foi estudado o kernel do Linux, versão 6.13.2, com o propósito de compreender melhor seu funcionamento interno e suas principais estruturas de dados, em especial aquelas que viabilizam o funcionamento do VFS.

O foco principal foi entender a arquitetura, as abstrações e as interfaces fornecidas pelo VFS, de forma a capacitar os estudantes para uma futura implementação e integração com um sistema de arquivos acadêmico em processo de desenvolvimento. Essa preparação envolveu o estudo dos mecanismos, interações e funcionamento de inodes, dentries, files, superblocks, assim como a operação de montagem de sistemas de arquivos no kernel Linux.

Também foi conduzido, com igual relevância, o estudo necessário para se alcançar uma compreensão completa do VFS. Esse esforço incluiu a análise das estruturas de dados oferecidas pelo kernel, o entendimento do gerenciamento e escalonamento de processos, bem como o funcionamento das system calls no contexto do Linux. Complementarmente, foi realizada uma leitura introdutória dos capítulos 1 e 2 do livro Linux Device Driver Development [Madieu 2022], com o objetivo de fornecer uma visão geral do ambiente de desenvolvimento de módulos no kernel.

1.1. Revisão Bibliográfica

Durante o período da AOE, foram conduzidas atividades voltadas ao estudo técnico e aplicado do kernel Linux, por meio da leitura de capítulos selecionados do livro Linux Kernel Development [Love 2010]. A atividade principal consistiu em uma leitura dirigida com reuniões semanais para discussões, acompanhamento e atividades práticas quando cabíveis. O foco esteve em compreender os fundamentos internos do kernel, como forma de preparação para o posterior aprofundamento na camada do Virtual Filesystem (VFS).

2. Conceitos Fundamentais e Ambiente de Desenvolvimento

Como parte das atividades de preparação e contextualização do trabalho com o Virtual Filesystem (VFS), foi realizada a leitura dos capítulos 1 e 2 do livro *Linux Device Driver Development*. Essa leitura teve como objetivo fornecer uma base técnica para o entendimento do ambiente de desenvolvimento no kernel Linux, incluindo conceitos de compilação, versionamento, configuração do kernel e estrutura dos módulos.

Arquiteturas de Compilação

Foram introduzidos os conceitos de *host* e *target*, distinguindo a máquina onde o processo de build ocorre (*host*) da máquina onde os binários gerados serão executados (*target*). A compilação pode ser:

- Nativa (*native compilation*): *host* e *target* são a mesma máquina.
- Cruzada (*cross-compilation*): *host* e *target* são diferentes, exigindo o uso de um compilador cruzado (*cross-compiler*).

Controle de Versionamento do Kernel

O kernel Linux passou por diferentes esquemas de versionamento ao longo do tempo:

- **Antes de 2003**: esquema par/ímpar.
- **Entre 2003 e 2011**: esquema semantic versioning (X.Y.Z).
- **Após 2011**: esquema simplificado com versões X.Y arbitrárias.

O desenvolvimento é público e estruturado em repositórios como o *mainline* (de Linus Torvalds) e o *stable tree* (mantido com correções).

Toolchain e Configuração do Kernel

Para compilar o kernel é necessário um *toolchain*, que inclui ferramentas como *binutils*, compiladores (ex: gcc, clang) e bibliotecas. A configuração do kernel é altamente parametrizável, incluindo opções booleanas, triestados (n, y, m) e parâmetros de tipos diversos (strings, inteiros, hexadecimais).

Módulos do Kernel

Todo driver de dispositivo no Linux é implementado como um módulo do kernel, mas nem todo módulo é um driver. Os módulos podem ser:

- ***Built-in***: compilados diretamente no kernel, disponíveis desde o *boot*, mas não podem ser descarregados.
- ***Loadable***: compilados separadamente e carregados dinamicamente durante a execução, exigem um sistema de arquivos.

Os pontos de entrada e saída de um módulo são definidos pelas funções `module_init()` e `module_exit()`, respectivamente, sendo esta última necessária apenas para módulos dinâmicos.

Processos de *Build*: *In-Tree* e *Out-of-Tree*

- ***In-Tree Build***: o módulo faz parte do código-fonte principal do kernel e segue seu fluxo de compilação.
- ***Out-of-Tree Build***: o módulo é desenvolvido e compilado separadamente.

Reflexões Iniciais

A leitura também provocou questionamentos relevantes para os próximos passos do projeto, como: o módulo a ser desenvolvido será *built-in* ou dinâmico (*loadable*)? Será desenvolvido como parte interna do kernel (*in-tree*) ou de forma independente (*out-of-tree*)?

3. Fundamentos do Kernel Linux e Sistemas Operacionais Unix-like

Como parte do embasamento teórico inicial, foi realizada a leitura dos capítulos 1 e 2 do livro Linux Kernel Development [Love 2010], com o objetivo de compreender os conceitos fundamentais sobre o funcionamento do kernel Linux e seu contexto dentro dos sistemas operacionais derivados ou inspirados no Unix. Essa leitura forneceu uma visão geral sobre a estrutura e os papéis do kernel, bem como suas principais diferenças em relação a aplicações em espaço de usuário (*user-space*).

Unix e *Linux*: Características e Origens

O *Unix* se destaca historicamente por sua simplicidade, modularidade e portabilidade. Algumas de suas principais características, herdadas e adaptadas pelo Linux, incluem:

- A filosofia de que ”tudo é um arquivo”.
- A escrita do kernel em linguagem C.
- Criação rápida de processos com o uso de `fork()`.
- Mecanismos de comunicação entre processos (IPC) simples, porém robustos.

O Linux surgiu em 1991, criado por Linus Torvalds, motivado pela ausência de um sistema Unix gratuito e flexível. Embora inspirado no Unix, o Linux foi escrito do zero e não compartilha código com o Unix original, constituindo-se como um sistema operacional independente e de código aberto.

Espaços de Execução: *kernel-space* e *user-space*

O *kernel* Linux opera em um modelo de separação entre dois espaços de execução:

- ***User-space***: onde são executadas aplicações e processos iniciados pelo usuário.
- ***Kernel-space***: onde está o código do *kernel*, incluindo *drivers*, gerenciadores de memória, escalonadores e manipuladores de interrupções.

Os processadores, em qualquer instante, estão executando um dos três estados possíveis:

- Em ***user-space***, executando código de um processo.
- Em ***kernel-space***, no contexto de um processo (ex: uma *system call*).
- Em ***kernel-space***, em contexto de interrupção, sem associação a processos.

System Calls e Interrupções

Aplicações comunicam-se com o *kernel* por meio de chamadas de sistema (*system calls*). Quando uma *system call* é executada, o *kernel* age em nome do processo chamador, entrando em *kernel mode* com acesso total ao *hardware* e à memória protegida.

Já interrupções ocorrem quando dispositivos de *hardware* demandam atenção do sistema. Essas interrupções são tratadas por funções específicas chamadas *interrupt handlers*, que são executadas em um contexto especial, fora do contexto de processos.

Características Específicas do *kernel* Linux

O *kernel* do Linux possui características distintas em comparação com aplicações de espaço de usuário:

- Ausência de biblioteca C (`libc`): O *kernel* não é vinculado à `libc` nem a bibliotecas padrão, visando performance e redução de tamanho.
- Sem proteção de memória: O *kernel* possui acesso direto e irrestrito à memória. Erros podem comprometer todo o sistema.
- Evita uso de ponto flutuante: Operações de ponto flutuante exigem gerenciamento explícito de registradores, o que é custoso e arriscado.
- Pilha pequena e fixa: O tamanho da pilha do *kernel* é limitado e depende da arquitetura (tipicamente 4 KB ou 8 KB no x86), sendo configurado em tempo de compilação.

Versões do *kernel* Linux

O esquema de versionamento do *kernel* segue o formato:

Major.*Minor*.Revisão(e opcionalmente uma versão estável)

Esse formato permite distinguir lançamentos principais, versões de desenvolvimento e versões com correções estáveis.

4. Gerenciamento de Processos e Escalonamento no Kernel Linux

A leitura dos capítulos 3 e 4 do livro Linux Kernel Development [Love 2010] aprofundou o entendimento sobre o modelo de processos no Linux, a estrutura interna do *kernel* para gerenciamento de processos e as estratégias de escalonamento utilizadas. Esta base é essencial para compreender como o *kernel* aloca recursos de CPU e mantém a responsividade do sistema mesmo em ambientes concorrentes.

Processo: Definição e Estrutura

Um processo é definido como um programa em execução, associado a recursos como arquivos abertos, sinais pendentes, memória virtual e, eventualmente, múltiplas *threads*. No Linux, processos e *threads* são representados por uma estrutura comum, diferenciando-se apenas por *flags* internas.

Criação e Execução de Processos

A criação de processos no Linux segue o modelo tradicional *Unix* de `fork()` seguido, opcionalmente, de `exec()`. O `fork()` duplica o processo atual, criando um novo com um identificador distinto (PID). A chamada `exec()` substitui o conteúdo do processo filho por um novo programa. Para otimizar esse processo, o kernel implementa o mecanismo de *Copy-On-Write* (COW), que evita a cópia imediata da memória do processo pai até que uma escrita efetivamente ocorra.

Término e Hierarquia de Processos

Um processo finaliza sua execução por meio da *system call* `exit()`, que libera seus recursos e notifica o *kernel*. A hierarquia entre processos é mantida a partir do processo `init` (PID 1), iniciado ao final do processo de *boot*. Cada processo possui um pai, e o *kernel* mantém ponteiros para o pai e para seus filhos por meio da estrutura `task_struct`. Caso o pai de um processo finalize antes do filho, o *kernel* realiza o reparenteamento, atribuindo um novo processo como pai — frequentemente o próprio `init`.

Representação Interna: `task_struct` e `thread_info`

O *kernel* armazena os processos em uma lista circular duplamente encadeada, composta por descritores de processos (`struct task_struct`). Essa estrutura armazena todas as informações do processo, como estado, prioridade e estatísticas de escalonamento. Complementarmente, a estrutura `thread_info`, localizada no final da pilha do processo, contém um ponteiro para o `task_struct` correspondente.

Para otimizar alocações frequentes de estruturas como `task_struct` ou `inode`, o *kernel* utiliza o *slab allocator*, um sistema de gerenciamento de memória com caches especializados para tipos de objetos.

Estados de Execução de Processos

Um processo pode estar em diferentes estados, tais como:

- `TASK_RUNNING`: em execução ou pronto para executar.
- `TASK_INTERRUPTIBLE`: bloqueado, mas responde a sinais.
- `TASK_UNINTERRUPTIBLE`: bloqueado e insensível a sinais.
- `__TASK_TRACED`: sendo rastreado (debugging).
- `__TASK_STOPPED`: execução pausada.

Escalonador (*Scheduler*)

O subsistema de escalonamento do *kernel* é responsável por decidir qual processo será executado a seguir, por quanto tempo e com qual prioridade. Ele é essencial para balancear o uso da CPU entre processos com diferentes necessidades, como:

- *I/O-bound*: dependem de dispositivos de entrada/saída, exigem resposta rápida.
- *CPU-bound*: realizam processamento intenso, requerem maior tempo de CPU.

Evolução do Escalonador no Linux

Ao longo das versões, o Linux evoluiu seus algoritmos de escalonamento:

- Linux 1.x–2.4: escalonador simples e pouco escalável.
- O(1) *Scheduler* (**2.5**): melhora a escalabilidade, mas com baixa responsividade em desktops.
- *Completely Fair Scheduler* (CFS) (**2.6.23**): algoritmo balanceado com foco em justiça e equidade no uso da CPU.

Completely Fair Scheduler (CFS)

O CFS introduziu uma abordagem inovadora, onde o ”tempo”atribuído a cada processo é proporcional ao seu peso (definido pelo valor de `nice`). Diferente de um *timeslice* fixo, o CFS busca garantir justiça aproximada: todos os processos devem progredir proporcionalmente ao longo do tempo. O tempo mínimo configurável é tipicamente 1 ms.

Internamente, o CFS organiza os processos em uma árvore Rubro-Negra (*Red-Black Tree*), garantindo ordenação eficiente. As estatísticas de escalonamento são armazenadas na estrutura `sched_entity`, contida em cada `task_struct`. A função `schedule()` é o ponto de entrada do subsistema de escalonamento e gerencia todas as classes de processos.

Políticas e Prioridades de Escalonamento

O sistema de escalonamento utiliza diferentes políticas, com base em prioridades:

- *Nice*: valores entre -20 (alta prioridade) a 19 (baixa).
- *Real-time*: prioridades entre 0 e 99.

O tempo de execução antes da preempção é chamado de *timeslice* ou *quantum*, que precisa ser balanceado entre responsividade e *overhead*.

Além do CFS, o *kernel* permite o uso de outras políticas de escalonamento especializadas, implementadas por meio de *scheduler classes*, para lidar com diferentes tipos de cargas de trabalho.

5. Gerenciamento de Memória com *Slab Allocator* e Processos Zumbis

Nesta etapa do estudo, foi realizada uma investigação mais aprofundada sobre o gerenciamento de memória no *kernel* Linux, com foco no funcionamento do *Slab Allocator*, além da análise do conceito e impacto de processos zumbis.

Slab Allocator

O *Slab Allocator* é um mecanismo de alocação de memória baseado em caches de objetos, voltado para otimizar a criação e destruição frequente de estruturas do *kernel*. Ele permite a reutilização eficiente de objetos de tamanho fixo, minimizando a fragmentação e o *overhead* de alocação.

Estrutura do Sistema

Cada tipo de objeto gerenciado pelo *Slab Allocator* é associado a um cache específico, representado pela estrutura `kmem_cache`. Exemplos típicos incluem:

- `task_struct_cache`: para processos
- `inode_cache`: para inodes
- `file_cache`: para arquivos

Esses caches contêm blocos de memória chamados *slabs*, que armazenam múltiplas instâncias do mesmo tipo de objeto.

Componentes de um `kmem_cache`

Uma estrutura `kmem_cache` contém:

- Tamanho do objeto: em bytes.
- Nome do cache: descritivo, como `dentry_cache`.
- Funções auxiliares: como construtores e destruidores.
- Alinhamento: para otimização em cache da CPU.
- Listas de *slabs*: organizadas em `free`, `partial` e `full`.
- *Flags* e estatísticas: controle de comportamento e contadores de uso.

Organização e Alocação de Objetos

Cada *slab* possui campos como:

- `list`: encadeamento entre *slabs*.
- `s_mem`: início da área de objetos.
- `inuse`: número de objetos ativos.
- `freelist`: controle de *slots* livres.

A versão moderna do *Slab Allocator*, chamada *SLUB*, elimina a lista `freelist` separada e utiliza ponteiros dentro dos próprios objetos para formar a lista livre, economizando memória.

Fluxo de Alocação e Liberação

- Alocação: o *kernel* procura um *slab* com espaço livre, aloca um objeto e atualiza os ponteiros. Caso não encontre, cria um novo *slab* com o *buddy allocator*.
- Liberação: o objeto é devolvido ao cache. Se o *slab* ficar completamente livre, ele pode ser mantido para uso futuro ou liberado, conforme heurísticas internas.

Vantagens do *Slab Allocator*

- Redução da fragmentação: a reutilização de objetos evita ”buracos”na memória.
- Eficiência para tipos fixos: ótimo desempenho para estruturas como `task_struct`, `inode`, `dentry`.
- Economia de tempo e recursos: evita alocações e liberações frequentes com `kmalloc`.

Comparação com `kmalloc`

A principal diferença entre `kmalloc` e o *Slab Allocator* está no tipo de uso. Enquanto `kmalloc` é uma função genérica usada para alocar blocos de memória de tamanho arbitrário, o *Slab Allocator* é otimizado para a reutilização de objetos fixos, frequentemente utilizados no kernel.

Quanto ao tamanho da alocação, `kmalloc` permite solicitar quantidades arbitrárias de memória, o que o torna mais flexível. Por outro lado, o *Slab Allocator* trabalha com tamanhos fixos por tipo de cache, o que o torna mais eficiente quando os mesmos tipos de objetos são frequentemente alocados e desalocados.

No aspecto de reutilização, `kmalloc` possui uma capacidade menor, já que não é focado em reaproveitamento — cada alocação tende a ser nova. O Slab Allocator, em contraste, apresenta alta reutilização, pois mantém objetos previamente alocados prontos para reutilização, reduzindo *overhead*.

Processos Zumbis

Um processo zumbi é um processo que terminou sua execução, mas cuja entrada na tabela de processos ainda não foi removida, pois o processo pai não coletou seu código de saída. Essa condição representa um estágio intermediário de término.

Como um Processo Zumbi é Criado

1. O processo pai cria um filho com `fork()`.
2. O filho termina sua execução (normalmente ou por erro).
3. O status de saída do filho é mantido até que o pai execute `wait()` ou `waitpid()`.
4. Enquanto isso não acontece, o processo permanece como zumbi.

Impactos dos Processos Zumbis

Um dos principais problemas causados por processos zumbis é o consumo de recursos. Cada processo, mesmo após sua finalização, mantém uma entrada na tabela de processos até que seu pai leia seu status. Como essa tabela possui um número limitado de entradas, a presença de muitos zumbis pode eventualmente impedir a criação de novos processos, afetando o funcionamento do sistema.

Além disso, a existência de zumbis costuma ser um forte indício de erro de programação. Isso geralmente significa que o processo pai não está gerenciando corretamente seus filhos, ou seja, não está realizando a chamada `wait()` (ou equivalente) para coletar o status de término dos processos filhos. Esse descuido pode comprometer a robustez e a estabilidade do software.

O que Acontece se o Pai Morre Antes?

Se o processo pai termina antes do filho, o *kernel* automaticamente designa o processo `init` (PID 1) como novo pai. Esse processo é responsável por coletar os filhos órfãos, evitando a criação de zumbis.

6. System Calls no Kernel Linux

As *system calls* (chamadas de sistema) são o principal mecanismo de comunicação entre processos em espaço de usuário (*user-space*) e o núcleo do sistema operacional (*kernel-space*). Elas fornecem uma interface controlada para que aplicações possam acessar recursos protegidos do sistema, como arquivos, dispositivos, memória e processos.

Função das *System Calls*

As *system calls* exercem três funções centrais:

- **Abstração do hardware:** permitem que aplicações interajam com dispositivos sem conhecer detalhes específicos.
- **Segurança e estabilidade:** controlam o acesso aos recursos conforme permissões do processo.
- **Virtualização:** garantem execução segura e isolada de múltiplos processos, inclusive com memória virtual.

No Linux, todas as interações entre um programa e o *kernel* são feitas exclusivamente por meio de *system calls*.

System Calls e POSIX

O Linux segue a especificação POSIX, que define padrões de comportamento para chamadas de sistema em sistemas operacionais tipo Unix. As aplicações geralmente não fazem chamadas diretas às *system calls*, mas sim às funções da biblioteca C (libc), que por sua vez realizam a chamada apropriada ao *kernel*.

Funcionamento e Retorno

As chamadas de sistema são invocadas por funções da libc que recebem parâmetros, causam efeitos colaterais e retornam um valor do tipo `long`. A convenção de retorno é:

- Valor zero (0): indica sucesso (em alguns casos).
- Valor negativo: indica erro; o código do erro pode ser obtido pela variável global `errno`.

A função `perror()` pode ser usada para exibir uma mensagem legível baseada no valor de `errno`.

Implementação no Kernel

Cada *system call* no kernel possui:

- Um número único (índice na tabela de chamadas `sys_call_table`).
- Um nome padronizado (ex.: `sys_getpid()`).
- Uma função associada, geralmente com a diretiva `asmlinkage`, que força que os argumentos sejam passados pela pilha.

System calls inválidas ou removidas retornam `-ENOSYS` por meio da função `sys_ni_syscall()`.

Mecanismo de Execução

Para executar uma system call, um processo deve seguir alguns passos. Primeiro, ele precisa carregar o número da chamada de sistema em um registrador apropriado, como o registrador eax no caso da arquitetura x86. Em seguida, o processo deve gerar uma interrupção de software, como a instrução int $0x80 em sistemas x86, ou usar a instrução `syscall` em sistemas x86_64.

Após isso, o kernel assume o controle em modo privilegiado, consulta a estrutura conhecida como `sys_call_table` e, com base no número fornecido, executa a função correspondente à chamada de sistema solicitada.

Passagem de Parâmetros (x86-32)

Parâmetro	Registrador
1º	EBX
2º	ECX
3º	EDX
4º	ESI
5º	EDI
6º+	Ponteiro para estrutura com argumentos

Segurança e Verificações

Como as chamadas são executadas no contexto do *kernel*, é essencial validar os parâmetros — especialmente ponteiros vindos do espaço do usuário. Deve-se verificar se:

- O endereço está no espaço de memória do processo.
- As permissões de leitura/escrita são adequadas.
- O processo tem as capacidades necessárias (ex.: `capable(CAP_SYS_NICE)` para alterar `nice` de outros processos).

Adição de uma Nova *System Call*

Para adicionar uma nova *system call* ao *kernel*, é necessário seguir uma sequência de etapas. Primeiro, deve-se implementar a função desejada no código do *kernel*, normalmente no arquivo `kernel/sys.c`. Em seguida, é preciso registrar essa função na `sys_call_table`, para que o *kernel* possa reconhecê-la como uma chamada válida.

Também é necessário atribuir um número único à nova chamada no arquivo `asm/unistd.h`, correspondente à arquitetura em uso. Por fim, o *kernel* deve ser recompilado para que a nova *system call* esteja disponível no sistema.

Vantagens e Desvantagens de Criar uma *System Call*

Entre as vantagens de implementar uma funcionalidade como *system call*, destacam-se a interface direta com o *kernel*, a boa performance e a simplicidade de implementação.

Por outro lado, há algumas desvantagens importantes: esse tipo de solução pode ser difícil de manter fora da árvore principal do *kernel*, exige suporte explícito para todas as arquiteturas utilizadas e não pode ser acessada por *scripts* nem integrada diretamente ao sistema de arquivos.

Alternativas a uma System Call

Em vez de criar uma nova chamada de sistema, é possível usar outras interfaces:

- Device nodes: com uso de `read()`, `write()` e `ioctl()`.
- Sysfs (`/sys`): ideal para expor variáveis do *kernel*.
- Interfaces baseadas em arquivos: como uso de `/proc` para diagnósticos.

Comparativo com o *Windows*

No *Windows*, as chamadas de sistema não são documentadas nem estáveis entre versões. Elas são acessadas por meio de funções de baixo nível da `ntdll.dll`, como `NtReadFile()`, e não há suporte oficial para usá-las diretamente — o que torna o desenvolvimento com *syscalls* no *Windows* complexo e frágil.

7. Estruturas de Dados no Kernel Linux

O kernel do Linux adota estruturas de dados genéricas e reutilizáveis. Essas estruturas são otimizadas para o uso em espaço de kernel e integração com subsistemas. Desenvolvedores são incentivados a reutilizá-las ao invés de criar estruturas personalizadas. As principais estruturas de dados fornecidas pelo kernel incluem:

- Listas Ligadas (*Linked Lists*)
- Filas (*Queues*)
- Mapas (*Maps*)
- Árvores Binárias (*Binary Search Trees*)

Listas Ligadas

Listas ligadas são uma das estruturas de dados mais utilizadas no kernel Linux, devido à sua simplicidade e flexibilidade. Elas armazenam elementos (ou nós) que são conectados por ponteiros, permitindo inserções e remoções eficientes em qualquer posição da lista. Diferente de arrays, os elementos não ocupam posições contíguas na memória. As listas ligadas são implementadas por meio da estrutura `struct list_head`, definida em `<linux/list.h>`.

Um subtipo importante é a **lista circular**. Nesse modelo, o último elemento da lista aponta de volta para o primeiro, e vice-versa, eliminando ponteiros `NULL` e permitindo percursos contínuos e seguros mesmo em listas vazias.

Manipulação de Listas

O kernel fornece diversas funções inline para manipulação de listas:

- `list_add()`, `list_add_tail()` – inserção
- `list_del()`, `list_del_init()` – remoção
- `list_move()`, `list_splice()` – movimentação e junção
- `list_empty()` – verificar se está vazia

Em vez de transformar uma estrutura em lista, adiciona-se um campo `list_head` à estrutura e ela participa das operações da lista.

Filas (Queues)

Para o padrão produtor-consumidor, o kernel fornece a estrutura `kfifo`, uma fila circular implementada em `<linux/kfifo.h>`. Ela mantém dois ponteiros: `in` (escrita) e `out` (leitura).

Principais Operações com kfifo

- `kfifo_alloc()` – aloca fila
- `kfifo_in()`, `kfifo_out()` – enfileirar/desenfileirar dados
- `kfifo_out_peek()` – leitura sem remoção
- `kfifo_size()` – obtém tamanho do buffer

Mapas (Maps)

Mapas associam chaves a valores. No kernel, o tipo mais comum é a estrutura `idr`, que associa identificadores únicos (UIDs) a ponteiros.

Operações com `idr`

- `idr_init()`: inicializa a estrutura `idr`.
- `idr_destroy()`: libera todos os recursos associados.
- `idr_get_new()`: associa um ponteiro a um novo ID disponível.
- `idr_find()`: recupera o ponteiro associado a um ID.
- `idr_remove()`: remove a associação entre um ID e seu ponteiro.

Árvore Binária de Busca (BST)

Organiza dados de forma ordenada: nós à esquerda possuem valor menor; à direita, maior. BSTs balanceadas garantem que a profundidade das folhas não varie muito, mantendo a eficiência.

Red-Black Trees

Árvores rubro-negras (RBTrees) são BSTs auto-balanceadas que seguem as regras:

- Todo nó é vermelho ou preto
- Folhas são pretas e não armazenam dados
- Um nó vermelho nunca tem filhos vermelhos
- Todos os caminhos da raiz às folhas têm o mesmo número de nós pretos

`rbtree` é implementada em `lib/rbtree.c` com cabeçalho em `<linux/rbtree.h>`.

Uso no Kernel

O kernel fornece as estruturas `rb_root` e `rb_node`, além de funções auxiliares. A lógica de comparação e inserção deve ser implementada pelo desenvolvedor.

Outras Estruturas do Kernel

Além das estruturas mais tradicionais, como listas ligadas, filas e árvores binárias, o kernel do Linux faz uso de outras estruturas para resolver problemas específicos. Um exemplo são as radix trees (ou tries), utilizadas para armazenar dados indexados por chaves inteiras, como identificadores ou endereços de memória. Elas são especialmente úteis em contextos como a gestão do espaço de endereçamento virtual de processos, pois permitem buscas, inserções e remoções rápidas com baixo overhead.

Outra estrutura são os bitmaps, que representam conjuntos de bits em forma compacta. Bitmaps são frequentemente usados para rastrear recursos — como blocos de memória, CPUs disponíveis ou flags de controle — são eficientes nas operações de verificação e mudança no valor dos bits, ambas executadas com instruções de baixo nível.

8. *Virtual Filesystem* (VFS)

O ***VFS*** é a camada do kernel Linux que permite chamadas genéricas como `open()`, `read()` e `write()` funcionarem de forma independente do sistema de arquivos ou do meio físico subjacente. Isso possibilita que diferentes sistemas de arquivos coexistam e que arquivos sejam acessados e movimentados entre eles com as mesmas chamadas padrão, algo que não ocorria em sistemas antigos como DOS, onde o acesso a sistemas não nativos dependia de ferramentas específicas.

O VFS funciona como uma camada de abstração que define interfaces e estruturas de dados comuns que todos os sistemas de arquivos devem seguir. Cada sistema de arquivos adapta sua implementação para esse modelo, mantendo sua lógica interna oculta do restante do kernel. Por exemplo, quando um programa executa `write`, a chamada chega a uma função genérica (`sys_write()`), que então invoca o método específico do sistema de arquivos para realizar a escrita física.

Estruturas e Abstrações Unix

No modelo Unix, os sistemas de arquivos são estruturados a partir de quatro abstrações principais: arquivos, entradas de diretório (dentries), inodes e pontos de montagem (mount points). O sistema de arquivos é visto como uma hierarquia única e contínua, diferente dos sistemas Windows ou DOS que usam letras para unidades distintas.

Arquivos são sequências ordenadas de bytes, acessíveis via nomes, podendo ser lidos, escritos, criados e deletados. Diretórios, por sua vez, são arquivos que contêm outras entradas, como subdiretórios ou arquivos. Um caminho de arquivo, como `/home/daniel/test`, é composto por várias entradas de diretório chamadas *dentries*: `/`, `home`, `daniel` e `test`.

O **inode** é a estrutura que guarda os metadados do arquivo, como permissões, tamanho, dono e datas, separando essa informação do conteúdo real. Já o **superblock** contém dados de controle sobre todo o sistema de arquivos, como número total de inodes e status.

Abordagem Orientada a Objetos no VFS

O VFS usa uma abordagem orientada a objetos em C, organizando seus dados em estruturas que contêm dados e ponteiros para funções que operam sobre eles. Os principais objetos do VFS são:

- `superblock`: representa um sistema de arquivos montado;
- `inode`: representa um arquivo ou diretório;
- `dentry`: representa uma entrada de diretório, componente de um caminho;
- `file`: representa um arquivo aberto associado a um processo.

Não existe um objeto distinto para diretórios, que são tratados como arquivos comuns no VFS. Além disso, existem estruturas como `file_system_type` (representa um tipo de sistema de arquivos registrado), `vfsmount` (representa um ponto de montagem), `fs_struct` (contém informações do sistema de arquivos para um processo) e `files_struct` (lista os arquivos abertos por um processo).

Superblock

O objeto `superblock` armazena informações específicas do sistema de arquivos e é representado pela `struct super_block`. Ele possui um ponteiro `s_op` para uma tabela de operações (`struct super_operations`) que contém funções como `write_inode()` e `sync_fs()` para manipulação de baixo nível.

Inode

O objeto `inode` é representado pela `struct inode` e possui o ponteiro `i_op` para a tabela `inode_operations`, que define funções como `create()` para criar arquivos e `lookup()` para localizar arquivos dentro de diretórios.

Dentry

Os objetos `dentry` representam cada componente do caminho do arquivo e são criados dinamicamente pelo VFS. Eles atuam como uma ponte entre os nomes dos arquivos/diretórios e seus respectivos inodes, acelerando a resolução dos caminhos sem necessidade de acessar o disco repetidamente. O cache de dentries (`dcache`) armazena essas estruturas para otimizar o desempenho, evitando leituras repetidas.

Cada dentry pode estar em um dos três estados: usado (apontando para um inode válido com usuários), não usado (inode válido, mas sem usuários) ou negativo (sem inode associado).

O kernel usa uma tabela hash para organizar dentries, facilitando buscas rápidas. Quando um caminho é acessado pela primeira vez, seus dentries são criados e armazenados no cache; acessos posteriores são atendidos rapidamente por essa cache.

File

O objeto `file` representa arquivos abertos pelos processos, sendo criado na chamada `open()` e destruído em `close()`. Ele contém informações como modo de acesso e deslocamento atual (*offset*), e aponta para um dentry, que por sua vez aponta para um inode. Os usuários interagem diretamente com o objeto `file`, usando chamadas como `read()`, `write()` e `lseek()`.

Outras Estruturas Importantes

No VFS, outras estruturas importantes incluem o `file_system_type`, que representa o tipo de sistema de arquivos, como ext4 ou UDF, definindo suas características gerais.

O `vfsmount` corresponde a uma instância montada do sistema, gerenciando os pontos de montagem e sua relação na hierarquia de arquivos. Cada processo mantém uma `files_struct`, que contém a lista dos arquivos atualmente abertos por ele, enquanto a `fs_struct` armazena informações sobre o diretório de trabalho atual e o diretório raiz associados ao processo. Além disso, a `namespace` controla a hierarquia de sistemas de arquivos que o processo pode enxergar, permitindo isolamento e controle de visibilidade dos recursos de arquivos.

9. Conclusão

Ao longo deste trabalho, foi possível compreender de forma profunda a arquitetura do *kernel Linux*, especialmente no que diz respeito ao *Virtual Filesystem* (VFS). Por meio do estudo das principais estruturas de dados, como ***inode***, ***dentry***, ***file*** e ***superblock***, evidenciou-se como o VFS atua como uma camada de abstração poderosa e essencial para a interoperabilidade entre diferentes sistemas de arquivos. Esse entendimento teórico é fundamental para qualquer desenvolvedor que pretenda atuar na construção ou extensão de funcionalidades do *kernel*.

Além disso, a análise dos mecanismos de gerenciamento de processos e memória, como o escalonador CFS e o *Slab Allocator*, proporcionou uma visão abrangente sobre a complexidade e a eficiência das operações realizadas em nível de *kernel*. O estudo dos processos zumbis, *system calls* e a comparação entre estratégias de alocação de memória revelou práticas essenciais para garantir estabilidade, desempenho e segurança em sistemas operacionais modernos.

Por fim, a Atividade Orientada de Ensino foi uma oportunidade para consolidar conhecimento sobre o funcionamento interno do *kernel Linux* e também ampliar a capacidade analítica e crítica diante dos desafios de desenvolvimento no ambiente do *kernel Linux*. O conteúdo estudado servirá como alicerce para futuras iniciativas acadêmicas e profissionais, como a implementação de módulos e sistemas de arquivos personalizados. Assim, a atividade cumpriu seu papel formativo, preparando os envolvidos para interações mais seguras, eficientes e fundamentadas com o núcleo do sistema operacional.

Referências

Love, R. (2010). *Linux Kernel Development*. Pearson Education.

Madieu, J. (2022). *Linux Device Driver Development: Everything you need to start with device driver development for Linux kernel and embedded Linux*. Packt Publishing Ltd.