# KERNEL

# Conceitos

- **Kernel = Núcleo**
  Kernel de um programa como parte central, fundamental de um programa/algoritmo.
- **Kernel = Modo Kernel(Supervisor)**
  Parte de um programa que executa em modo Kernel. Modo Kernel -> Espaço/Modo de Execução. Suporte do Processador a diferentes espaços, ou niveis, de execução(modo Kernel e modo usuario).

# Conceitos

- POSIX

Portable Operating System Interface for UNIX. Um conjunto de padrões da IEEE e ISO que definem como programas e sistemas operacionais de interface com os outros. Sistemas operacionais compatíveis com POSIX incluem Windows NT e a maioria das versões do UNIX.

# Kernel de um SO

- Núcleo do Sistema Operacional -> Prove o Gerenciamento do Sistema e Funcionalidades Basicas.

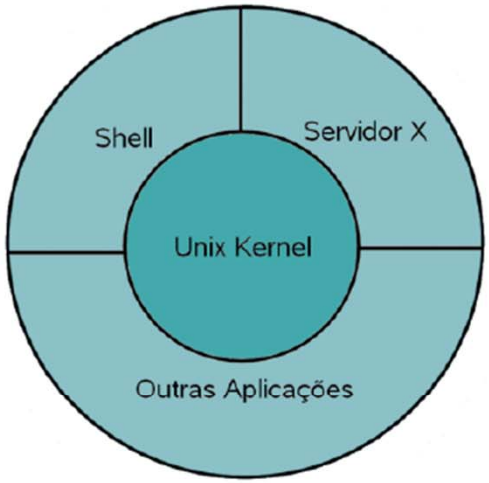



Figura: Diagrama do Sistema Operacional Unix

O kernel é a parte mais importante do sistema operacional, pois, sem ele, a cada programa novo que se criasse seria necessário que o programador se preocupasse em escrever as funções de entrada/saída, de impressão, entre outras, em baixo nível, causando uma duplicação de trabalho e uma perda enorme de tempo. Como o kernel já fornece a interface para que os programas possam acessar os recursos do sistema de um nível mais alto e de forma transparente, fica resolvido o problema da duplicação do trabalho.

# Ambientes de Execução

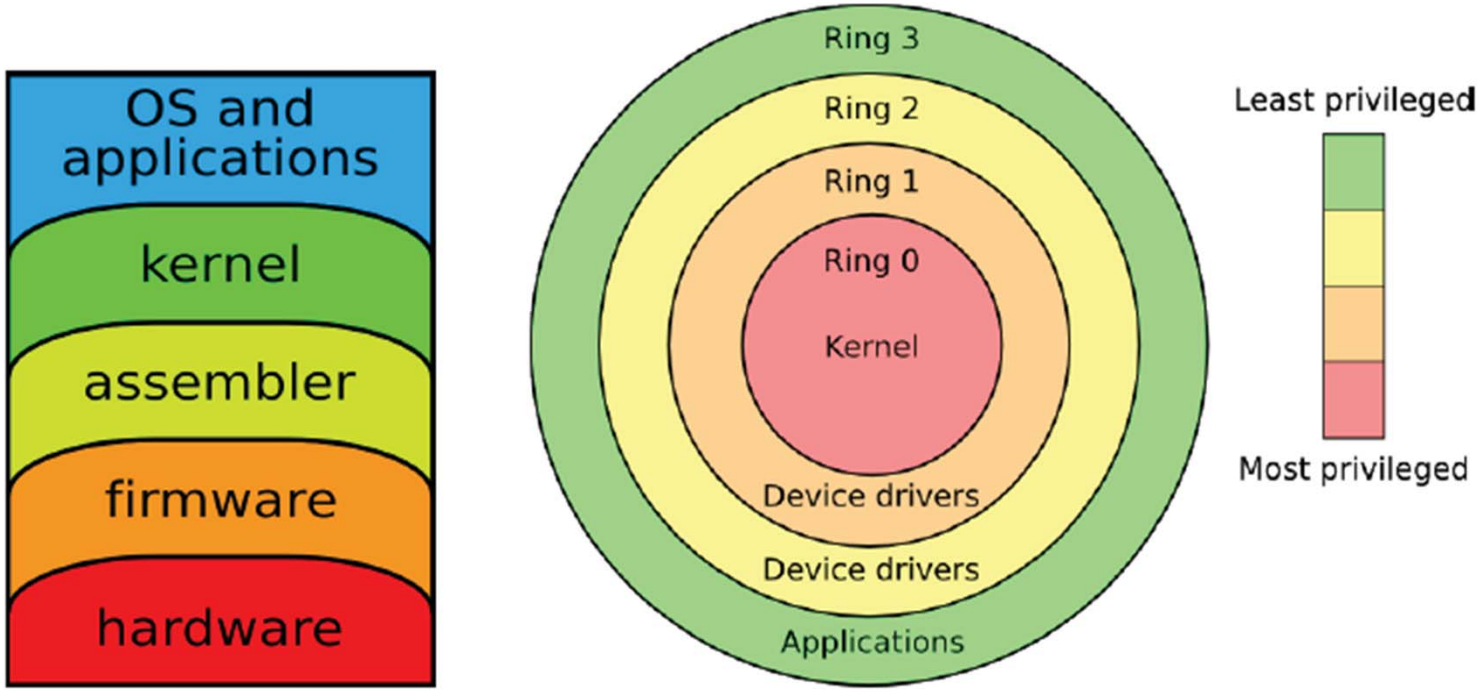



Figura: Diagramas de camadas de Abstração e Anéis de Privilégio de Execução

# IPC – Comunicação InterProcessos

A comunicação entre processos, em inglês Inter-Process Communication (IPC), é o grupo de mecanismos que permite aos processos transferirem informações entre si. A execução de um processo pressupõe por parte do sistema operacional, entre outras coisas, a criação de um contexto próprio de execução que, de certa forma, abstrai o processo dos componentes reais do sistema. Devido a esta virtualização dos recursos, o processo não tem conhecimento acerca dos outros processos e, como tal, não consegue trocar informações.

Desta forma, o kernel do sistema operacional, que tem acesso a toda a memoria disponível, irá atuar como canal de comunicação entre os processos. Para isto, existem diferentes mecanismos de IPC que entram em uso com base nos diferentes requisitos. As técnicas de IPC estão divididas em métodos de pipes, FIFOs, filas de mensagens, sincronização, memória compartilhada, semáforos e chamadas de procedimento remoto (RPC).

# IPC - Pipes

Um pipe é um mecanismo de fluxo unidirecional de dados (one-way flow) que permite que dois processos conexos (isto é, um é filho do outro) possam enviar uma sequência de bytes de um para outro. Naturalmente, para usar este tipo de mecanismo de forma adequada, é preciso formar uma espécie de protocolo no qual os dados são enviados através do pipe.

A ordem no qual os dados são gravados para o pipe é a mesma ordem em que os dados serão lidos. O sistema operacional garante que os dados não sejam perdidos, a menos que um dos processos saia repentinamente.

Pipes podem ser considerados arquivos abertos que não tem imagem correspondente no sistema de arquivos montado. Um pipe pode ser criado por meio da chamada de sistema pipe(), que retorna um par de descritores de arquivos. O processo pode ler a partir do pipe utilizando a chamada read() e, do mesmo modo, pode escrever no pipe usando a chamada write().

POSIX apenas define condutas half-duplex, por isso mesmo que a chamada de sistema pipe() retorna dois descritores de arquivos., cada processo deve fechar um antes de usar o outro. Se o houver a necessidade de um fluxo bidirecional de dados, os processos devem criar dois pipes diferentes, invocando a função pipe() duas vezes.

Vários sistemas do tipo UNIX, como o System V Release 4, implementa pipes full-duplex e permitem que ambos os descritores possam ser lidos ou escritos. Linux adota uma outra abordagem: cada descritor do pipe ainda é unidirecional, mas não é necessário fechar um deles antes de usar o outro.

# IPC - FIFO

FIFOs (first in, first out) são parecido com pipes. FIFOs também fornecem um fluxo de dados do tipo half-duplex como os pipes. A diferença entre FIFOs e pipes é que o primeiro é identificado no sistema de arquivos com um nome, enquanto o outro não é. Isto é, FIFOs são pipes. FIFOs são identificados por um ponto de acesso que é um arquivo no sistema de arquivos, ao passo que pipes são identificados por um ponto de acesso que é um inode. Outra grande diferença entre FIFOs e pipes é que FIFOs são permanentes, ou seja, estão criados enquanto o sistema operacional estiver funcionando, enquanto que os pipes duram apenas durante o ciclo de vida do processo em que foram criados. Uma vez que eles são identificados pelo sistema de arquivos, eles permanecem na hierarquia até que sejam explicitamente removidos utilizando o comando unlink.

Um processo cria um FIFO usando a chamada de sistema mknod(), passando o nome do caminho do novo FIFO e o valor S_IFIFO com a máscara de permissão do novo arquivo. POSIX introduz a chamada de sistema mkfifo() que cria um FIFO. Esta chamada é implementada em Linux, assim como em System V Release 4, para a biblioteca de funções C a função é conhecida por mknod(). Uma vez criado, um FIFO pode ser acessado pelas chamadas de sistema open(), read(), write() e close(). No entanto, o VFS trata de uma maneira especial porque o inode do FIFO e as operações dos arquivos são personalizadas e não dependem do sistema de arquivos em que o FIFO é armazenado.

# IPC - Mensagens

Processos podem sem comunicar por mensagens de IPC. Cada mensagem criada por um processo é enviado para uma lista de mensagens do IPC onde fica até que um processo a leia.
A mensagem é composta por um header (cabeçalho) de tamanho fixo e uma variável text de tamanho variável; que podem ser rotulados com um valor inteiro (o tipo de mensagem), que permite um processo de recuperar mensagens de uma fila. Uma vez que o processo tenha lido uma mensagem de uma fila, o kernel a destrói, portanto, apenas um processo pode receber uma dada mensagem.

Para se enviar uma mensagem, um processo chama a função msgsnd(), passando os seguintes parâmetros:

- O identificador do IPC da fila de mensagens do destino;
- O tamanho do texto;
- O endereço do buffer do modo usuário que contem o tipo de mensagem juntamente com o texto da mensagem.

Para receber uma mensagem, um processo invoca a função msgrcv(), passando:

- O identificador do IPC dos recursos da fila de mensagens do IPC;
- O apontador para o buffer do modo usuário de onde o tipo e o texto da mensagem devem ser copiados;
- O tamanho do buffer;
- Um valor t que especifica qual mensagem deve ser recebida.

# IPC – Memória Compartilhada

Memória compartilhada permite que um ou mais processos se comuniquem através da memória em todo o espaço de endereço virtual. As páginas da memória virtual são referenciadas por uma tabela de entradas de páginas, sendo que esta tabela é compartilhada com os processos.

Uma vez que a memória está sendo compartilhada, não existem controles sobre a forma como os processos estão acessando. Eles devem contar com outros mecanismos, por exemplo, semáforos ou até mesmo mensagens.

As principais funções que auxiliam no compartilhamento de memória são shmat() e shmdt(), onde a função shmat() agrega o segmento da memória compartilhada com o processo e, a função shmdt() desagrega o segmento do processo.

# Tipos de Kernel

- MicroKernel
  - NanoKernel
- Kernel Monolitico
- Kernel Hibrido*
- ExoKernel

*Kernel Hibrido é tido como um termo controverso dentre a comunidade ciêntifica da área.

# Abordagem de Design

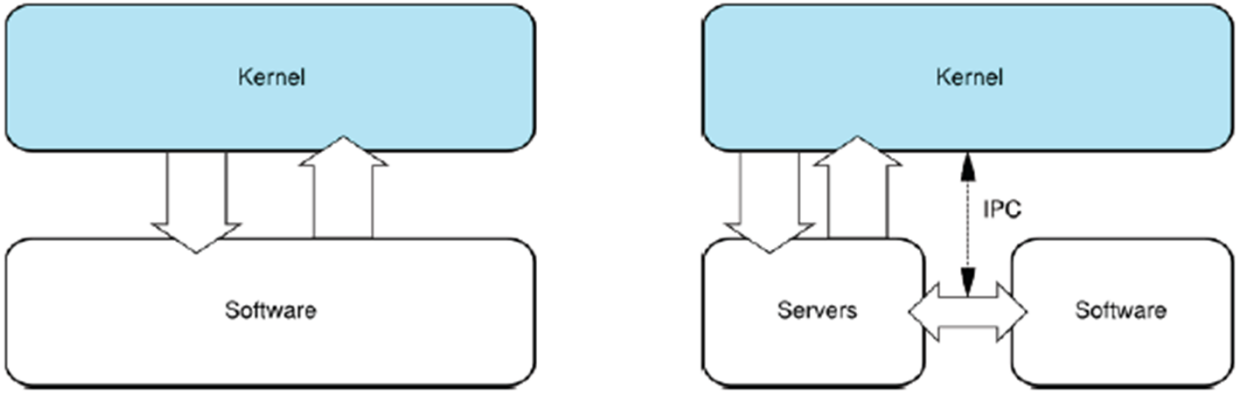



Figura: Diagramas Kernel Monolitico e MicroKernel

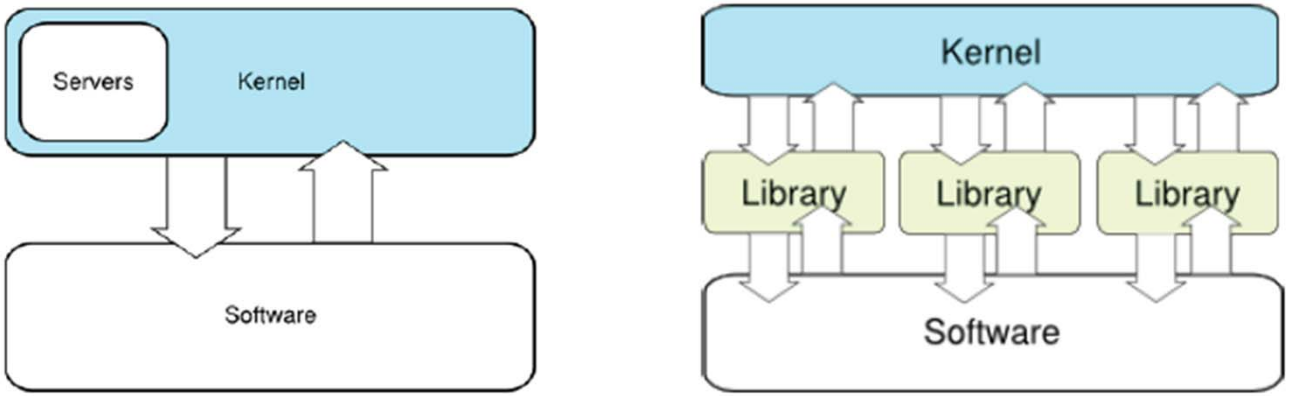



Figura: Diagramas Kernel Hibrido e ExoKernel

# MicroKernel

Os sistemas operacionais em estrutura cliente-servidor têm um microkernel, que é um kernel menor e mais simples. Para funcionar, esse sistema é dividido em processos. Existem processos clientes e processos servidores.

A proposta do microkernel é tornar o kernel menor e mais simples possível, movendo código para as camadas superiores. Como implementação, o sistema é dividido em processos, onde cada processo é responsável por oferecer um conjunto de serviços.

Sempre que uma aplicação necessita algum recurso ela solicita ao processo responsável. A aplicação que solicita um serviço é chamada de cliente, e o processo que atende a essa solicitação é chamado de servidor.

# MicroKernel

Nessa implementação, o sistema é dividido em processos, onde cada processo é responsável por oferecer um conjunto de serviços. Por exemplo:

- serviço de arquivos [ou servidor de arquivos];
- serviço de criação de processos [servidor de processos];
- serviço de memória [servidor de memória];
- serviço de rede [servidor de rede].

# MicroKernel

Num exemplo de funcionamento, se um processo de usuário [processo cliente] precisar ler um bloco de dados em arquivo, este fará uma solicitação ao serviço de arquivos que, por sua vez, irá acessar o sistema de arquivos, ler esse bloco de dados e enviar para o processo cliente.

Um microkernel permite que os processos servidores executem em modo usuário e não modo kernel [que é o caso do sistema monolítico]. Apenas o microkernel, que é o responsável pela comunicação entre clientes e servidores, executa no modo kernel.

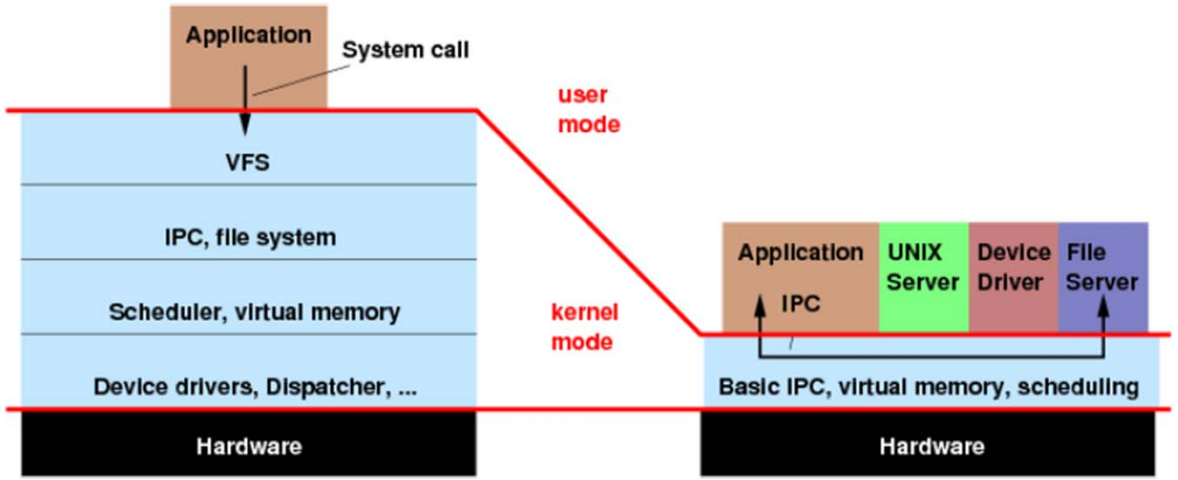
A função básica do microkernel é permitir a comunicação entre processos clientes e processos servidores. Essa comunicação ocorre pela troca de mensagens.

# MicroKernel

Uma vantagem é que o sistema operacional passa a ter manutenção mais simples, indiferente do fato do processo de serviço estar numa máquina com um único processador, com múltiplos processadores ou num sistema distribuído.

No entanto, a implementação do sistema cliente-servidor é difícil, então usualmente é implantado uma combinação de modelo em camadas com o cliente-servidor.

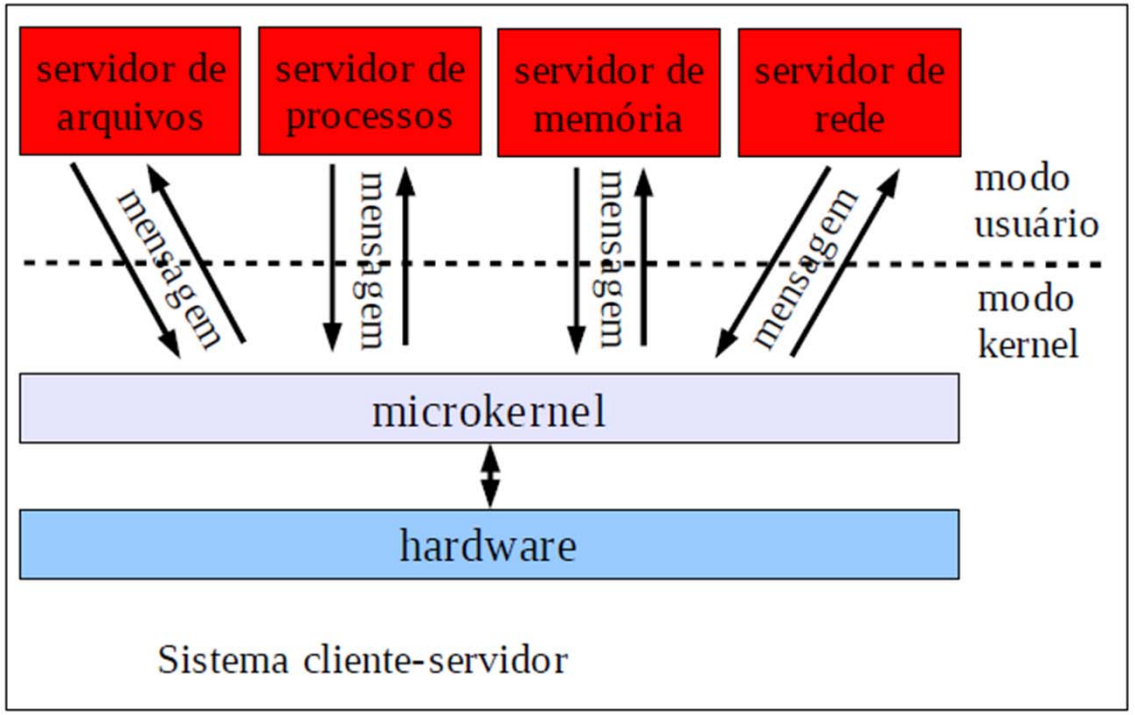
Ainda como desvantagem, tem baixo desempenho, pois é frequente a mudança do modo de acesso. Além diso, algumas funções exigem acesso ao hardware, então o microkernel precisa assumir também funções críticas como escalonamento, intercomunicação entre processos, gerência de memória, tratamento de interrupções e gerência de dispositivos.



# MicroKernel

Como vantagem, os serviços executam em modo usuário, então se algum falhar não comprometerá o sistema como um todo.

Também como vantagem, é facilmente adaptável para uso em sistemas tanto monoprocessados quanto multi, seja forte ou fracamente acoplado. Em sistema distribuídos, possibilita a um cliente solicitar um serviço e ser atendido remotamente, fato que garante alta escalabilidade [pode-se adicionar mais e mais máquinas].



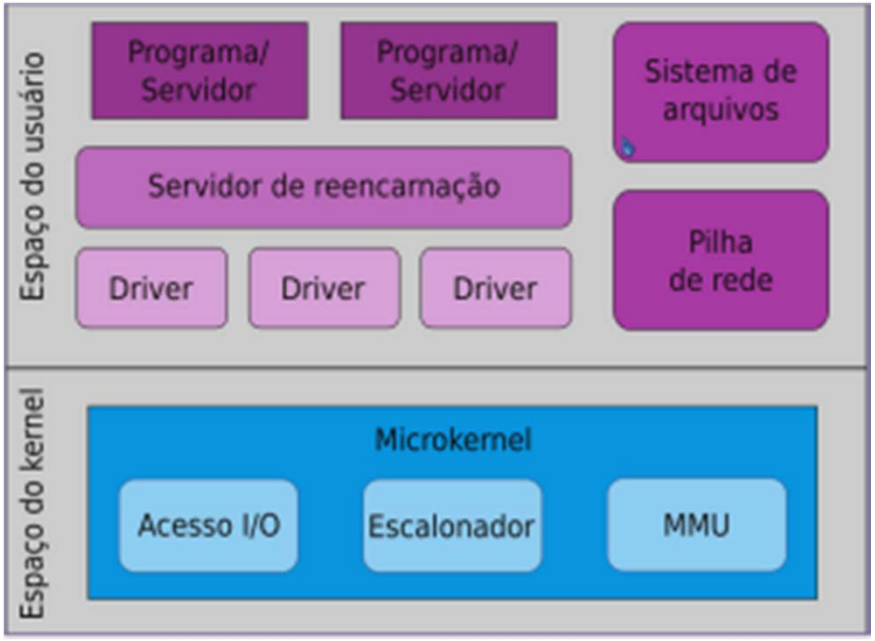
Sistema cliente-servidor

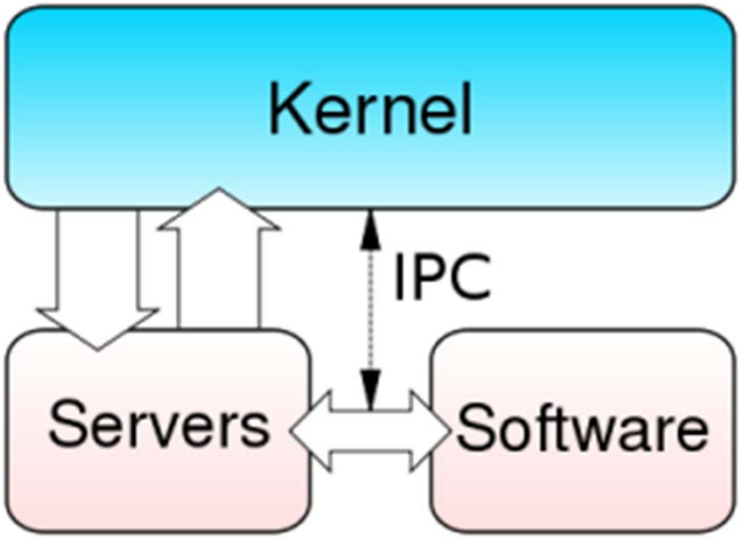
E ainda como vantagem, o sistema cliente-servidor é mais fácil de depurar pois os serviços são pequenos e específicos.

No passado, uma das razões para adoção do microkernel era a limitação de memória no hardware de 16 bits ou menos. Mas após o surgimento de arquiteturas de 32 bits, não houve mais necessidade do kernel ser pequeno.

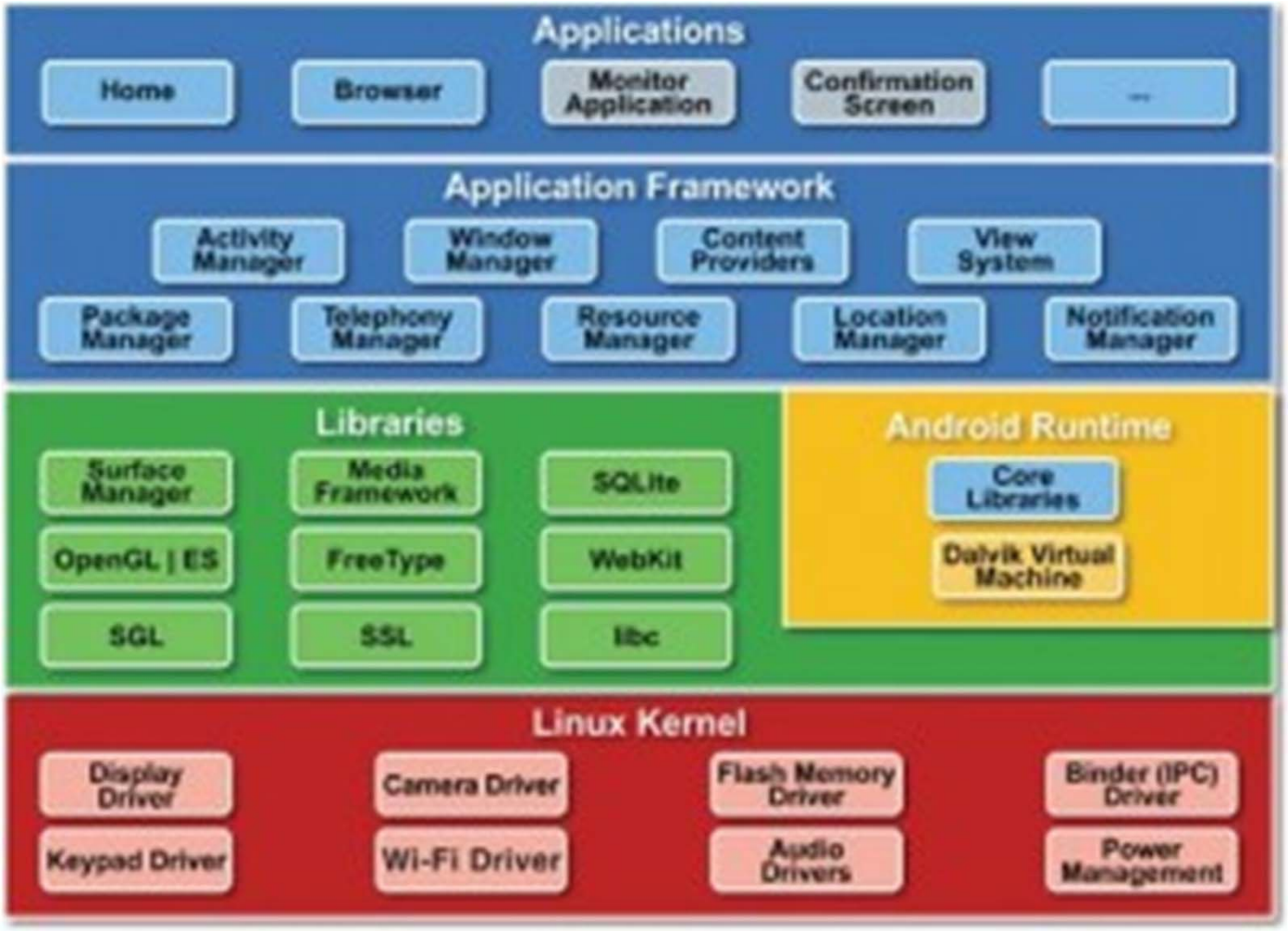
# MicroKernel

Hoje em dia o MicroKernel e usado em sistema embarcados pois existe a necessidade de uma aplicação não falhar, casos específicos exemplo é o roteador de alta performance da Cisco também usa um MicroKernel, ele detecta travamentos rapidamente e monitora continuamente a função de processos críticos, reiniciando processos travados conforme necessário, para manter o sistema em funcionamento. O MicroKernel puro não é usado em computação pessoal devido a baixa velocidade de operação.



# Exemplo de MicroKernel




LINUX ANDROID

# Kernel Monolítico

O modelo monolítico dominou nos primórdios da computação, nele o sistema operacional é escrito como um programa único composto por uma coleção de subrotinas que chamam umas às outras sempre que for necessário.

A organização mais comum é estruturar o sistema operacional como um conjunto de rotinas que podem interagir livremente umas com as outras. O sistema monolítico pode ser comparado a uma aplicação formada por várias rotinas ou procedimentos compilados separadamente e depois ligados [linked, "lincados"], para formar um grande e único programa executável.

# Kernel Monolítico

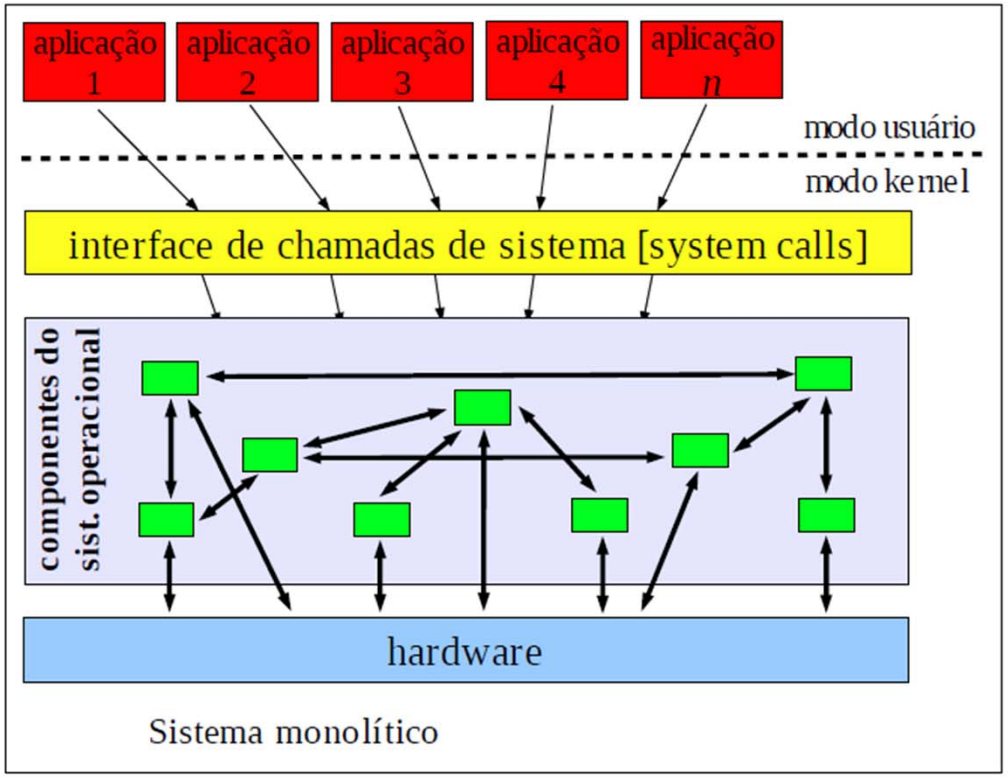
Como vantagem está o grande desempenho e a alta eficiência. Além disso, existe forte integração interna dos componentes e por isso permite que detalhes de baixo nível do hardware sejam bem explorados.

Como desvantagens estão manutenção e expansão complexos. Além disso, como os módulos tabalham no mesmo espaço de endereçamento, uma falha pode paralisar todo o kernel.

Os modelos monolíticos também adotam modo usuário e modo supervisor [kernel], que operam com diferentes privilégios e diferentes prioridades de execução. Às aplicações é reservado o modo usuário, com baixo privilégio e baixa prioridade, já para as rotinas do sistema operacional é reservado o modo supervisor.

# Kernel Monolítico

O sistema monolítico não é estruturado, mas também não é totalmente desestruturado. Por exemplo, existe um pouco de estrutura para os serviços do sistema que são requisitados via chamadas de sistema.



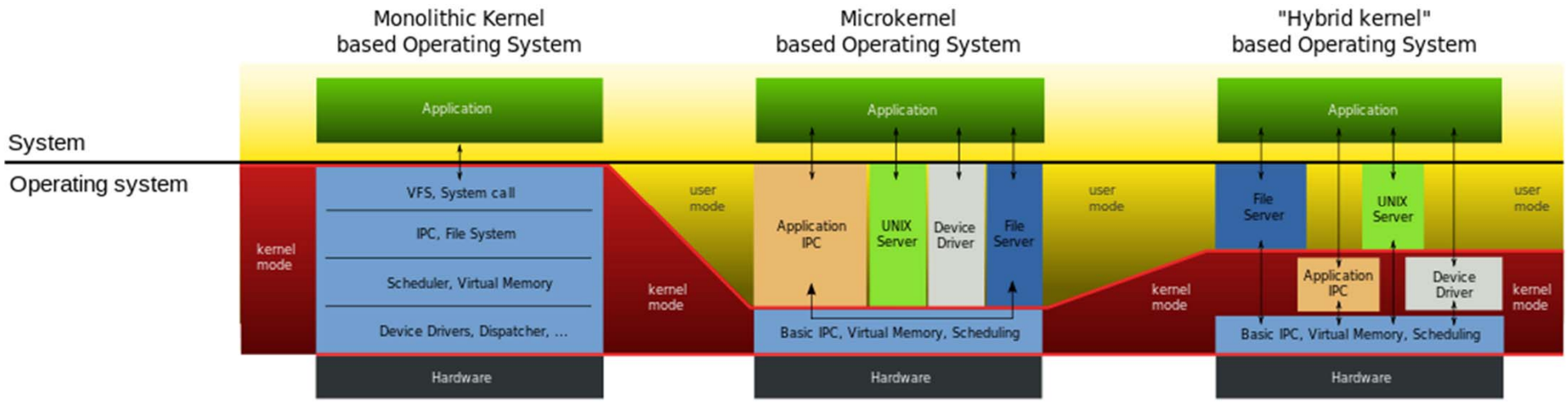
Sistema monolítico

O Linux é um exemplo de kernel monolítico e modular. Modular pois módulos podem ser carregados para acessar algum recurso, isso com o sistema rodando. Posteriormente, se não estiver mais em uso, esse módulo poderá ser descarregado para melhorar o desempenho da máquina e economizar memória.

Os primeiros Unix, no início dos anos 1970, também eram sistemas monolíticos. Atualmente, FreeBSD, AIX e HP-UX são sistemas monolíticos.

Outro exemplo é o antigo MS-DOS.

# Kernel Híbrido



# Exemplo de Kernel Híbrido

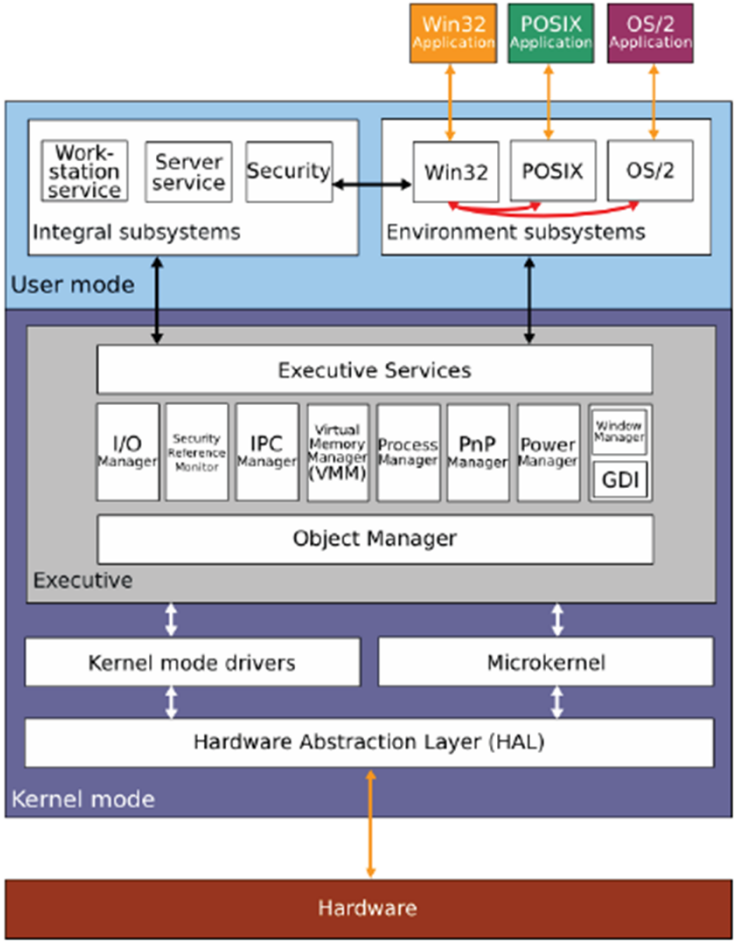



Figura: Diagrama do Kernel WindowsNT

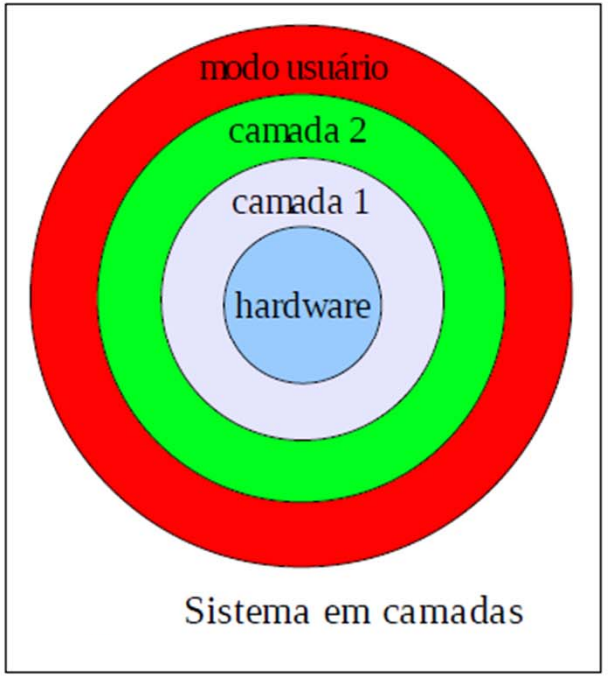
# Sistema em Camadas

No sistema em camadas, a comunicação ocorre apenas entre camadas adjacentes. Isso garante segurança e funcionamento para o sistema. A camada mais inferior é a que tem acesso aos dispositivos de hardware e a camada mais externa é a que realiza a interface com os usuários e com as aplicações.

A organização dos módulos ocorre por meio de hierarquia de camadas sobrepostas, onde cada módulo oferece um conjunto de funções que podem ser utilizados por outros módulos.

Os módulos de uma camada podem fazer referência apenas a módulos de camadas inferiores, e quanto mais interna a camada, mais privilegiada ela é.

A vantagem em estruturar o sistema operacional em camadas é a isolação que essas camadas proporcionam, pois facilitam a alteração e depuração do sistema e criam uma hierarquia de níveis de modos, que protege as camadas mais internas.



Sistema em camadas

# Sistema em Camadas

Porém, o empilhamento de várias camadas de software faz com que a requisição de uma aplicação demore mais tempo para chegar até o dispositivo ou recurso a ser acessado, prejudicando o desempenho do sistema. Além disso, não é tarefa óbvia dividir as funcionalidades de um kernel em camadas horizontais de abstração crescente, pois mesmo funcionalidades que tratam de recursos distintos podem ser interdependentes.

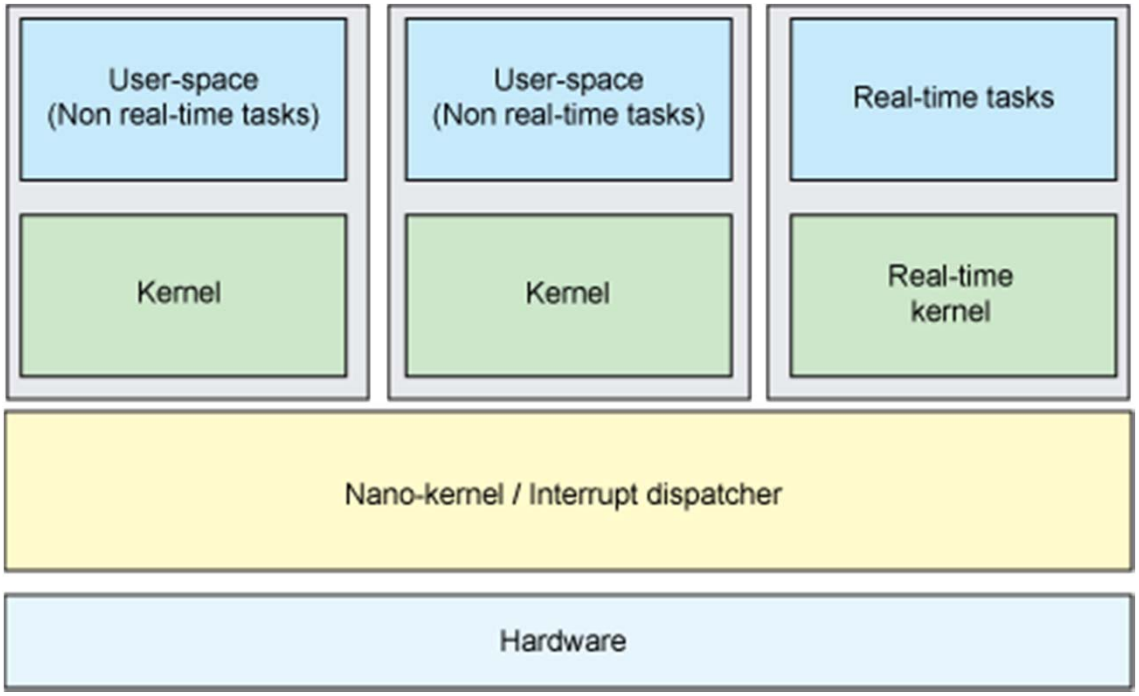
Muitos dos sistemas operacionais comerciais utilizam o modelo de duas camadas, onde uma camada trabalha no modo kernel e a outra no modo usuário.

Como exemplo, o MULTICS[1] tinha oito camadas.

Outro exemplo é o Minix, que foi criado pelo professor Andrew S. Tanenbaum em 1987. Atualmente o Minix é um projeto de código fonte aberto [http://www.minix3.org/].

# NanoKernel

Enquanto a abordagem thin kernel conta com um kernel minimizado que inclui gerenciamento de tarefas, a abordagem nano-kernel dá um passo adiante ao minimizar ainda mais o kernel. Passa, assim, a ser menos um kernel e mais uma Hardware Abstraction Layer (HAL). O nano-kernel fornece ao hardware o compartilhamento de recursos para vários sistemas operacionais funcionando em uma camada mais alta. Como o nano-kernel abstrai o hardware, pode fornecer uma priorização para sistemas operacionais de camadas mais altas e, portanto, oferecer suporte em tempo real hard.



# NanoKernel

Observe as semelhanças entre essa abordagem e a abordagem de virtualização para executar vários sistemas operacionais. Neste caso, o nano-kernel abstrai o hardware a partir dos kernels em tempo real e dos que não são em tempo real. Isto é similar à forma como os hypervisores abstraem o hardware vazio dos sistemas operacionais convidados. Um exemplo da abordagem nano-kernel é o Adaptive Domain Environment for Operating Systems (ADEOS). O ADEOS oferece suporte a vários sistemas operacionais concorrentes que estejam executando simultaneamente. Quando ocorrerem eventos de hardware, o ADEOS consulta cada sistema operacional de uma cadeia para verificar qual irá lidar com o evento.