

Sistemas Operacionais

Aula 02 - Estruturas do Sistema Operacional

Prof. Samuel Souza Brito

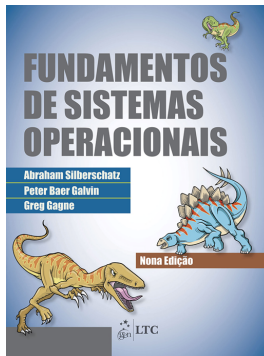
Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Aplicadas – ICEA
Departamento de Computação e Sistemas – DECSI



UFOP

João Monlevade-MG
2024/2

- O material aqui apresentado é baseado no Capítulo 2 do livro:
 - Silberschatz, A.; Galvin, P. B.; Gagne, G., Fundamentos de Sistemas Operacionais, Editora LTC, 9ª edição, 2015.

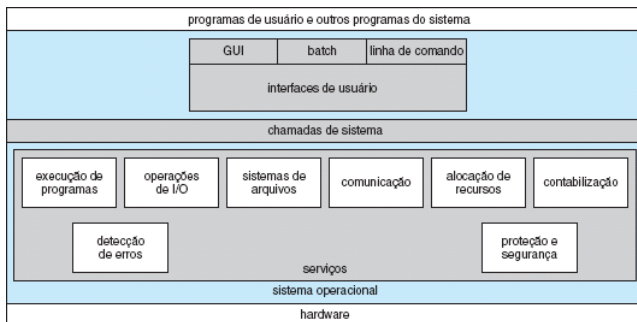


- SO fornece o ambiente dentro do qual os programas são executados.
- Internamente, variam muito em sua composição.
- O projeto de um novo SO é uma tarefa de peso.
 - É importante que os objetivos sejam bem definidos antes de o projeto começar.
- Podemos considerar um SO segundo vários critérios:
 - Serviços fornecidos
 - Interface disponibilizada para usuários e programadores
 - Componentes e suas interconexões

Serviços do Sistema Operacional

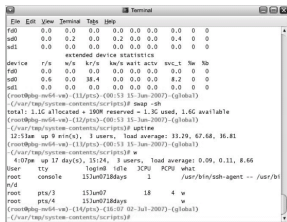
Serviços do Sistema Operacional

- SO fornece certos serviços para programas e para os usuários desses programas.
- Os serviços específicos fornecidos diferem de um SO para outro, mas podemos identificar classes comuns.



Interface de usuário

- Quase todos os SOs têm uma **interface de usuário (UI - *user interface*)**.
- Pode assumir várias formas:
 - **Interface de linha de comando (CLI - *command-line interface*)**
 - **Interface batch**
 - **Interface gráfica de usuário (GUI - *graphical user interface*)**
- Alguns sistemas fornecem duas dessas variações ou as três.



```
File Edit View Terminal Tabs Help
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
sd0 0.0 0.2 0.0 0.0 0.2 0.0 0.0 0.4 0.0 0.0 0.0
sdl 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
extended device statistics
device r/w w/s kr/s km/s wait actv svc_t tsw kb
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
sd0 0.6 0.0 38.4 0.0 0.0 0.0 0.0 8.2 0.0 0.0 0.0
sdl 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
(root@blg-m64-wa)~(13/pts)~(00:53 15-Jan-2007)~(global)
~/var/tmp/system-contents/scripts# wmap -dh
total: 1.1G allocated + 180M reserved = 1.3G used, 1.6G available
(root@blg-m64-wa)~(12/pts)~(00:53 15-Jan-2007)~(global)
~/var/tmp/system-contents/scripts# uptime
12:51am up 9 mins, 3 users, load average: 33.20, 67.68, 36.81
(root@blg-m64-wa)~(13/pts)~(00:53 15-Jan-2007)~(global)
~/var/tmp/system-contents/scripts# w
4:07pm up 17 day(s), 35124, 3 users, load average: 0.09, 0.11, 8.66
User tty login idle CPU what
root console 1531m0718days 1 /usr/bin/ssh-agent -- /usr/bi
m/d
root pts/3 1531m07 18 4 w
root pts/4 1531m0718days w
(root@blg-m64-wa)~(14/pts)~(16:07 02-Jul-2007)~(global)
~/var/tmp/system-contents/scripts#
```



- SO deve ser capaz de carregar um programa na memória e executar esse programa.
- O programa deve ser capaz de encerrar sua execução, normal ou anormalmente (indicando o erro).

- Um programa em execução pode requerer E/S.
 - Pode envolver um arquivo ou um dispositivo de E/S.
- Para dispositivos específicos, funções especiais podem ser desejáveis.
 - Como a gravação em um drive de CD ou DVD.
- Para eficiência e proteção, os usuários geralmente não podem controlar os dispositivos de E/S diretamente.
 - SO deve fornecer um meio para executar E/S.

- Programas precisam ler e gravar arquivos e diretórios.
 - Também precisam criar e excluí-los pelo nome, procurar um arquivo específico e listar informações de arquivos.
- Alguns SOs incluem o gerenciamento de permissões para permitir ou negar acesso a arquivos ou diretórios com base no proprietário dos arquivos.
- Muitos SOs fornecem uma variedade de sistemas de arquivos.
 - Para permitir a escolha pessoal e/ou fornecer recursos específicos ou características de desempenho.
 - NFS, NTFS, FAT, ext, entre outros.

- Um processo pode trocar informações com outro processo.
- Essa comunicação pode ocorrer entre processos sendo executados no mesmo computador ou entre processos sendo executados em sistemas de computação diferentes.
- As comunicações podem ser implementadas por:
 - **Memória compartilhada:** dois ou mais processos leem e gravam em uma seção compartilhada da memória.
 - **Troca de mensagens:** pacotes de informações em formatos predefinidos são transmitidos entre processos pelo SO.

- SO precisa detectar e corrigir erros constantemente.
- Erros podem ocorrer no hardware da CPU e da memória, em dispositivos de E/S e no programa do usuário.
- Para cada tipo de erro, o SO deve tomar a medida apropriada para assegurar a computação correta e consistente.
- Em algumas situações: interromper o sistema.
- Em outras: encerrar um processo causador de erro ou retornar um código de erro ao processo para que este detecte o erro e o corrija.

- Os serviços apresentados anteriormente são fornecidos pelo SO para auxiliar o usuário.
- Existe outro conjunto de funções cujo objetivo é assegurar a operação eficiente do próprio sistema.
 - Veremos alguns deles a seguir.

- Quando existem múltiplos usuários ou jobs ativos ao mesmo tempo, é necessário alocar recursos para cada um deles.
- SO gerencia muitos tipos diferentes de recursos.
 - Alguns podem ter um código especial de alocação, como os ciclos de CPU, a memória principal e o armazenamento em arquivos.
 - Outros podem ter um código muito mais genérico de solicitação e liberação, como os dispositivos de E/S.

- Controlar quais usuários utilizam que quantidade e que tipos de recursos do computador.
- Essa monitoração pode ser usada a título de contabilização (para que os usuários possam ser cobrados) ou para acumulação de estatísticas de uso.
 - As estatísticas de uso podem ser uma ferramenta valiosa para pesquisadores que desejem reconfigurar o sistema para melhorar os serviços de computação.

- Os proprietários de informações armazenadas em um sistema de computação multiusuário ou em rede podem querer controlar o uso dessas informações.
- Quando vários processos separados são executados concorrentemente, um processo não pode interferir nos outros ou no próprio SO.
- **Proteção:** garantir que qualquer acesso a recursos do sistema seja controlado.
- **Segurança** do sistema contra invasores também é importante.
 - Exigência de que cada usuário se autentique junto ao sistema para obter acesso aos recursos do sistema.
 - Defesa de dispositivos externos de E/S, incluindo adaptadores de rede, contra tentativas de acesso ilegal, e à gravação de todas essas conexões para a detecção de invasões.

Chamadas de Sistema

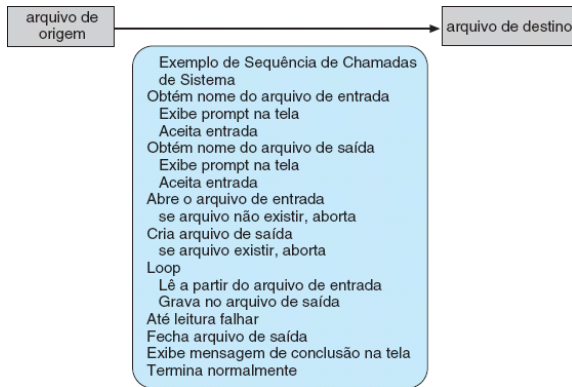
- **Chamadas de sistema (system calls)** fornecem uma interface com os serviços disponibilizados por um SO.
- Estão disponíveis como rotinas escritas em C e C++.
 - Certas tarefas de nível mais baixo podem ter que ser escritas em linguagem assembly.
 - Tarefas onde o hardware precisa ser acessado diretamente.
- UNIX/Linux:
 - `#include <unistd.h>`

Chamadas de Sistema – Exemplo

- Programa que lê os dados de um arquivo e copia esses dados para um novo arquivo:

Chamadas de Sistema – Exemplo

- Programa que lê os dados de um arquivo e copia esses dados para um novo arquivo:



- Mesmo programas simples podem fazer uso intenso do SO.
- Frequentemente, os sistemas executam milhares de chamadas de sistema por segundo.
 - A maioria dos programadores nunca vê esse nível de detalhe.
- Desenvolvedores de aplicações projetam programas de acordo com uma **interface de programação de aplicações (API - application programming interface)**.
 - Especifica um conjunto de funções que estão disponíveis para um programador de aplicações, incluindo os parâmetros que são passados a cada função e os valores de retorno que o programador pode esperar.
 - API Windows (Win32), API Java¹, API POSIX, etc.

¹

Exemplo: <https://docs.oracle.com/javase/8/docs/api/>

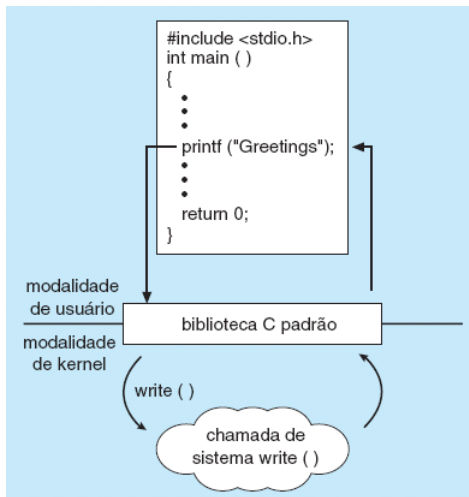
- Por que usar uma API em vez de invocar chamadas de sistema reais?

- Por que usar uma API em vez de invocar chamadas de sistema reais?
 - Portabilidade de programas.

- Por que usar uma API em vez de invocar chamadas de sistema reais?
 - Portabilidade de programas.
 - Chamadas de sistema são mais detalhadas e difíceis de manipular.
 - `exemplo1.cpp`, `exemplo2.cpp` e `exemplo3.cpp` (disponível no Moodle).

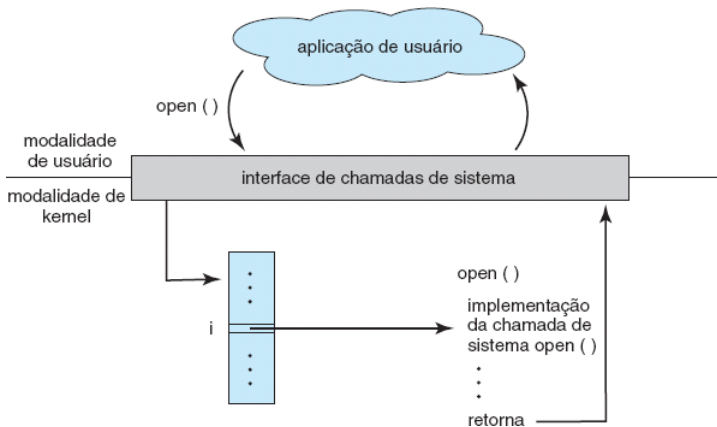
Chamadas de Sistema

- Programa em C invocando a função `printf()`, que utiliza a chamada de sistema `write()`:



- Quem usa uma chamada de sistema não precisa ter qualquer informação sobre como ela foi implementada.
 - Precisa apenas conhecer a interface e saber o que o SO fará como resultado da chamada.
 - A maioria dos detalhes ficam escondidos dos usuários atrás de APIs.
- Tipicamente, cada chamada de sistema possui um número associado.
 - A interface de chamada de sistemas mantém uma tabela indexada com esses números.
 - A interface de chamada de sistema invoca a chamada desejada no kernel do SO e retorna o estado e valores de retorno para o usuário.

Relação entre API e Chamada de Sistema

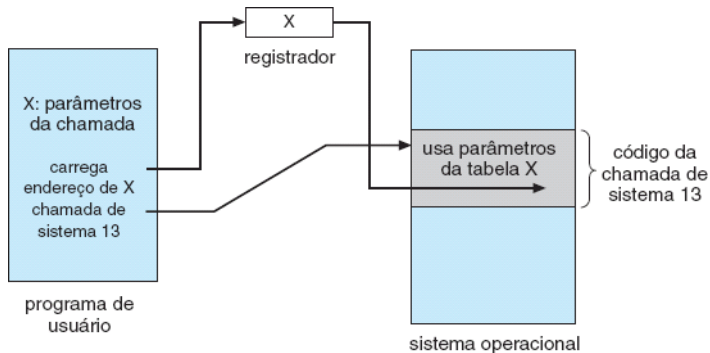


- A execução de uma chamada de sistema requer mais do que a identificação da chamada desejada.
 - Para obter entradas, por exemplo, podemos ter que especificar o arquivo ou dispositivo a ser usado como origem, entre outras informações.

- Três métodos são comumente utilizados para se passar parâmetros para o SO:
 - Passagem por meio de registradores da CPU.
 - Armazenamento em um bloco (tabela) na memória cujo endereço é passado em um registrador da CPU.
 - Linux e Solaris.
 - Armazenamento na pilha, de onde o SO extrai os dados.
- Mais comuns: método do bloco ou da pilha.
 - Por quê?

- Três métodos são comumente utilizados para se passar parâmetros para o SO:
 - Passagem por meio de registradores da CPU.
 - Armazenamento em um bloco (tabela) na memória cujo endereço é passado em um registrador da CPU.
 - Linux e Solaris.
 - Armazenamento na pilha, de onde o SO extrai os dados.
- Mais comuns: método do bloco ou da pilha.
 - Por quê?
 - Não limitam o número ou tamanho dos parâmetros!

Passagem de parâmetros como uma tabela



Tipos de Chamadas de Sistema

Tipos de Chamadas de Sistema

- Seis categorias principais:
 - Controle de processos
 - Manipulação de arquivos
 - Manipulação de dispositivos
 - Manutenção de informações
 - Comunicações
 - Proteção

Exemplos de chamadas de sistema do Windows e UNIX

	Windows	Unix
Controle de Processos	CreateProcess () ExitProcess () WaitForSingleObject ()	fork () exit () wait ()
Manipulação de Arquivos	CreateFile () ReadFile () WriteFile () CloseHandle ()	open () read () write () close ()
Manipulação de Dispositivos	SetConsoleMode () ReadConsole () WriteConsole ()	ioctl () read () write ()
Manutenção de Informações	GetCurrentProcessID () SetTimer () Sleep ()	getpid () alarm () sleep ()
Comunicações	CreatePipe () CreateFileMapping () MapViewOfFile ()	pipe () shm_open () mmap ()
Proteção	SetFileSecurity () InitializeSecurityDescriptor () SetSecurityDescriptorGroup ()	chmod () umask () chown ()

- Um processo precisa ser capaz de interromper sua execução normalmente ou de forma anormal.
- Se terminar de forma anormal ou se encontrar um problema e causar uma exceção:
 - Pode ocorrer um despejo da memória e a geração de uma mensagem de erro.
 - O despejo é gravado em disco e pode ser examinado por um depurador para determinar a causa do problema.

- Um processo pode precisar de carregar e executar outro programa.
 - Exemplo: linha de comando.
- SO deve gerenciar a execução dos programas.
 - Listar e alterar atributos de um processo.
 - Criar e terminar um processo.
 - Alocar e liberar memória.
- Finalização de um processo:
 - Espera por tempo
 - Espera por evento
- Processos podem compartilhar dados.
 - Trancar (*lock*) e liberar dados compartilhados.

- Suporte a criação e deleção de arquivos.
 - Nome e uma série de atributos.
- Para que possa ocorrer a leitura/escrita/reposicionamento, arquivos precisam ser inicialmente abertos.
- Arquivos precisam ser fechados ao fim do seu uso.
- Ler e modificar atributos de arquivos.
- Mesmos conceitos se aplicam aos diretórios.

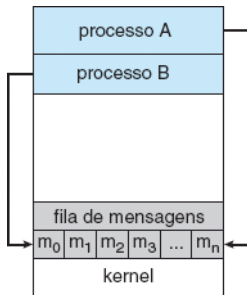
- Um programa pode precisar de recursos extras (memória, acesso a arquivos, etc) para executar.
 - Se os recursos estão disponíveis o programa poderá utilizá-los de imediato.
 - Caso contrário, será necessário aguardá-los.
- Recursos podem ser enxergados como sendo dispositivos físicos (unidades de disco, por exemplo) ou virtuais (arquivos, por exemplo).

- Em sistemas multiusuários, a requisição pelo dispositivo se faz necessária.
 - Obtenção de uso exclusivo.
 - Liberação do dispositivo.
 - Leitura, escrita e reposicionamento.
 - Semelhante as chamadas de sistema para arquivos.
- SOs podem tratar arquivos e dispositivos da mesma forma, por meio de um conjunto de chamadas de sistema.
 - Exemplo: Diretório `/dev` de sistemas Linux contém ponteiros para dispositivos de HW.

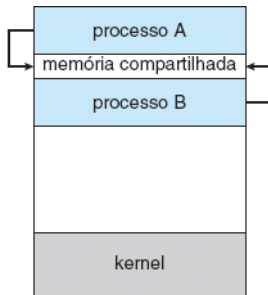
- Algumas chamadas de sistema são utilizadas somente para transferência de informações entre o programa de usuário e o SO.
 - Exemplos:
 - Retornar a hora e data atuais.
 - Retornar informações sobre o sistema.
- Outras chamadas de sistema são úteis para depuração de um programa.
 - Despejo de memória.
 - Rastreamento (*trace*) de programa.

- Chamadas de sistema para fornecimento de perfil de tempo de um programa.
 - Indica por quanto tempo ele é executado em uma locação específica ou conjunto de locações.
 - Exemplo: *gprof* (disponível no Moodle).
- Além disso, o SO mantém informações sobre todos os processos.
 - Chamadas de sistema para acessar e alterar essas informações.
 - Diretório `/proc` em sistemas Linux.

- Comunicação entre processos:
 - **Troca de mensagens:**
 - Por meio de um canal de comunicação.
 - **Memória compartilhada:**
 - Acesso a regiões de memória de outros processos.



(a)



(b)

- Proporciona um mecanismo para o controle de acesso aos recursos fornecidos por um sistema de computação.
- Historicamente, a proteção era uma preocupação somente em sistemas de computação multiprogramados com vários usuários.
 - Advento das redes e da Internet: todos os sistemas de computação devem se preocupar com a proteção.
- Chamadas de sistema que fornecem proteção incluem:
 - `set_permission()` e `get_permission()`: manipulam as definições de permissões para recursos (arquivos e discos, por exemplo).
 - `allow_user()` e `deny_user()`: especificam se determinados usuários podem (ou não) obter acesso a certos recursos.

Estrutura do Sistema Operacional

- SO moderno deve ser construído cuidadosamente para funcionar de maneira apropriada e ser facilmente modificável.
- Abordagem comum:
 - Divisão da tarefa em componentes pequenos, ou módulos, em vez da criação de um sistema monolítico.
- Diferentes organizações internas:
 - Simples
 - Camadas
 - Microkernels
 - Módulos

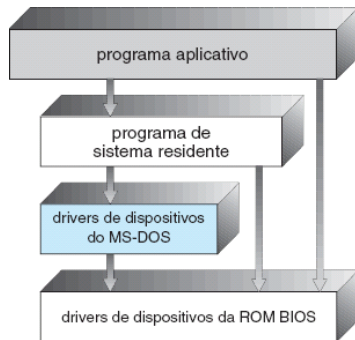
Estrutura Simples

- Muitos SOs não têm estruturas bem definidas.
 - Começam como sistemas pequenos, simples e limitados e, então, crescem para além de seu escopo original.

MS-DOS

- Originalmente projetado e implementado por algumas pessoas que não tinham ideia de que se tornaria tão popular.
- Escrito para fornecer o máximo de funcionalidade no menor espaço.
 - Não foi dividido cuidadosamente em módulos.
- Interfaces e níveis de funcionalidade não estão bem separados.
- Programas aplicativos podem acessar as rotinas básicas de E/S para gravar diretamente em tela e drives de disco.
 - Vulnerável a programas incorretos (ou maliciosos).
- Também ficou limitado em razão do hardware de sua época.
 - Intel 8088 não fornece modalidade dual e proteção de hardware.

Estrutura do MS-DOS:

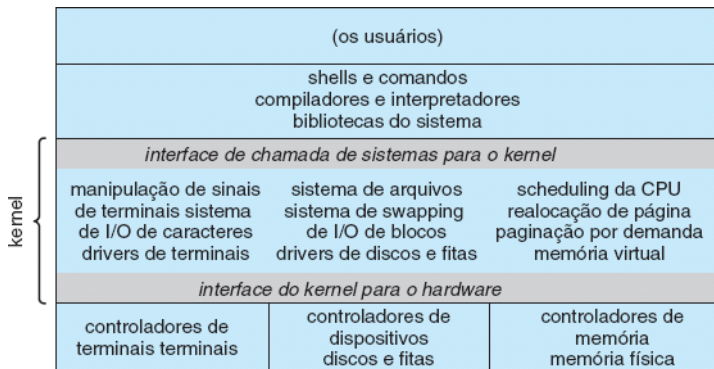


UNIX

- Inicialmente, foi limitado pela funcionalidade do hardware.
- Composto por duas partes separadas:
 - Kernel
 - Programas de sistema
- Kernel é separado em uma série de interfaces e drivers de dispositivos que foram sendo adicionados e expandidos com o passar dos anos.
- Estrutura monolítica difícil de implementar e manter.

Estrutura Simples

■ Estrutura do UNIX:

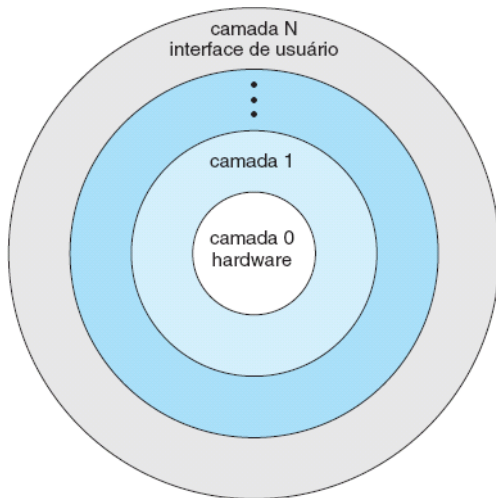


- Ainda vemos evidências dessa estrutura simples e monolítica em UNIX, Linux e Windows.

- SO é dividido em várias camadas (níveis).
 - Camada inferior (camada 0) é o hardware.
 - Camada mais alta (camada N) é a interface de usuário.
- Uma camada típica de SO (camada M , por exemplo) é composta por estruturas de dados e um conjunto de rotinas que podem ser invocadas por camadas de níveis mais altos.
 - Camada M pode invocar operações em camadas de níveis mais baixos.

Abordagem em Camadas

SO em camadas:



■ Vantagens:

■ **Vantagens:**

- Com a modularidade, cada nível se torna mais fácil de construir, usando as funções dos níveis inferiores.
- Facilidade de depuração.

■ **Dificuldade:**

■ Vantagens:

- Com a modularidade, cada nível se torna mais fácil de construir, usando as funções dos níveis inferiores.
- Facilidade de depuração.

■ Dificuldade:

- Definição apropriada das diversas camadas.
- Uma camada pode usar somente camadas de nível mais baixo.
 - Necessário um planejamento cuidadoso.
- Exemplo: em qual camada deve ficar o driver de dispositivos do *backing store* (espaço em disco usado por algoritmos de memória virtual)?

■ Vantagens:

- Com a modularidade, cada nível se torna mais fácil de construir, usando as funções dos níveis inferiores.
- Facilidade de depuração.

■ Dificuldade:

- Definição apropriada das diversas camadas.
- Uma camada pode usar somente camadas de nível mais baixo.
 - Necessário um planejamento cuidadoso.
- Exemplo: em qual camada deve ficar o driver de dispositivos do *backing store* (espaço em disco usado por algoritmos de memória virtual)?
 - Deve estar em um nível mais baixo do que as rotinas de gerenciamento da memória.
 - Gerenciamento da memória requer o uso do *backing store*.

- **Desvantagem:**

■ Desvantagem:

- Tende a ser menos eficiente do que outras abordagens.
- Cada camada adiciona *overhead* à chamada de sistema.
- O resultado final é uma chamada de sistema que demora mais do que em um sistema não estruturado em camadas.

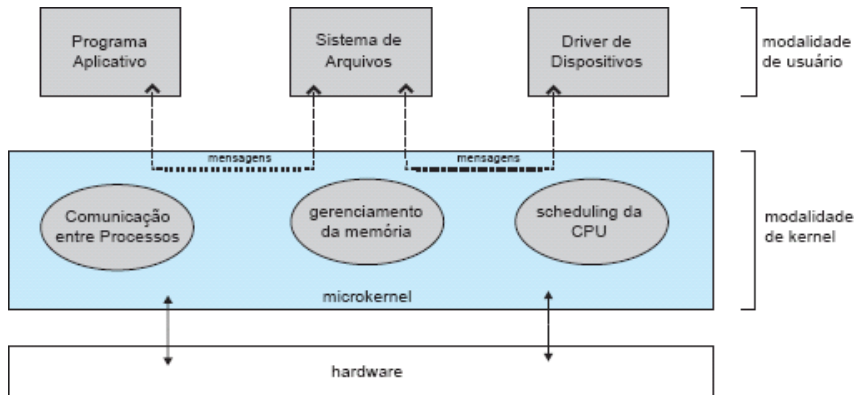
■ Desvantagem:

- Tende a ser menos eficiente do que outras abordagens.
 - Cada camada adiciona *overhead* à chamada de sistema.
 - O resultado final é uma chamada de sistema que demora mais do que em um sistema não estruturado em camadas.
- Atualmente, menos camadas com mais funcionalidades estão sendo projetadas.
- Fornecendo a maioria das vantagens do código modularizado, mas evitando os problemas de definição e interação de camadas.

- Estrutura o SO removendo todos os componentes não essenciais do *kernel*.
 - Implementando-os como programas de nível de sistema e de usuário.
- **Resultado:** *kernel* menor.
- Pouco consenso sobre quais serviços devem permanecer no *kernel* e quais devem ser implementados no espaço do usuário.
 - Microkernels normalmente fornecem um gerenciamento mínimo dos processos e da memória, além de um recurso de comunicação.

Microkernels

Arquitetura de um microkernel típico:



- Principal função do microkernel: fornecer comunicação entre o programa cliente e os diversos serviços que também estão sendo executados no espaço do usuário.
 - Comunicação é fornecida por transmissão de mensagens.
 - Exemplo: se o programa cliente deseja acessar um arquivo, ele deve interagir com o servidor de arquivos.
 - Programa cliente e o serviço nunca interagem diretamente.
 - Em vez disso, eles se comunicam indiretamente trocando mensagens com o microkernel.

- **Benefícios:**

■ Benefícios:

- Facilidade de extensão do SO.
 - Novos serviços são acrescentados ao espaço de usuário.
 - Não exigem modificação do kernel.

■ Benefícios:

- Facilidade de extensão do SO.
 - Novos serviços são acrescentados ao espaço de usuário.
 - Não exigem modificação do kernel.
- Mudanças no kernel tendem a ser menores.

■ Benefícios:

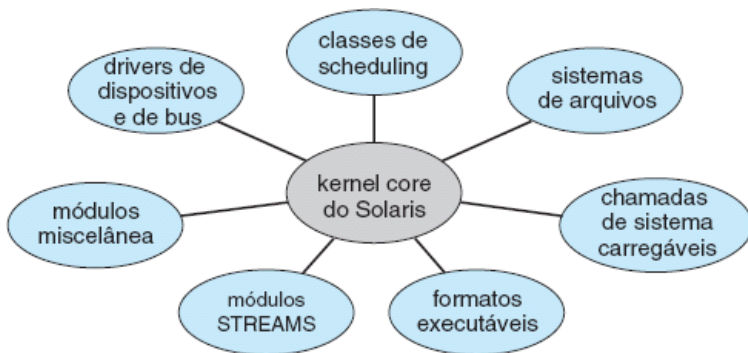
- Facilidade de extensão do SO.
 - Novos serviços são acrescentados ao espaço de usuário.
 - Não exigem modificação do kernel.
- Mudanças no kernel tendem a ser menores.
- Mais segurança e confiabilidade:
 - A maioria dos serviços executam como processos do usuário.
 - Se um serviço falhar, o restante do sistema permanecerá intocável.

■ Benefícios:

- Facilidade de extensão do SO.
 - Novos serviços são acrescentados ao espaço de usuário.
 - Não exigem modificação do kernel.
- Mudanças no kernel tendem a ser menores.
- Mais segurança e confiabilidade:
 - A maioria dos serviços executam como processos do usuário.
 - Se um serviço falhar, o restante do sistema permanecerá intocável.
- Tru64 UNIX, Mac OS X (Darwin), QNX, Windows NT (1ª versão).

- Módulos de *kernel* carregáveis.
- *Kernel* tem um conjunto de componentes nucleares e vincula serviços adicionais por meio de módulos.
 - Tanto em tempo de inicialização quanto em tempo de execução.
- Comum em implementações modernas do UNIX, Solaris, Linux, Mac OS X e Windows.
- Melhor vincular serviços dinamicamente do que adicionar novos recursos diretamente ao *kernel*.
 - Evita a recompilação do *kernel*.

■ Módulos carregáveis do Solaris:



- Linux também usa módulos do kernel carregáveis, principalmente no suporte a drivers de dispositivos e sistemas de arquivos.

- Poucos SOs adotam uma estrutura única rigidamente definida.
 - Combinam diferentes estruturas, resultando em sistemas híbridos que resolvem problemas de desempenho, segurança e usabilidade.

- Poucos SOs adotam uma estrutura única rigidamente definida.
 - Combinam diferentes estruturas, resultando em sistemas híbridos que resolvem problemas de desempenho, segurança e usabilidade.
- Linux e Solaris são monolíticos porque o desempenho é muito mais eficiente quando o SO ocupa um único espaço de endereçamento.
 - Eles também são modulares para que novas funcionalidades possam ser adicionadas ao *kernel* dinamicamente.

- Poucos SOs adotam uma estrutura única rigidamente definida.
 - Combinam diferentes estruturas, resultando em sistemas híbridos que resolvem problemas de desempenho, segurança e usabilidade.
- Linux e Solaris são monolíticos porque o desempenho é muito mais eficiente quando o SO ocupa um único espaço de endereçamento.
 - Eles também são modulares para que novas funcionalidades possam ser adicionadas ao *kernel* dinamicamente.
- Windows também é amplamente monolítico (por questões de desempenho, principalmente).
 - Mas retém certo comportamento típico de sistemas de microkernel.
 - Também fornecem suporte para módulos do kernel carregáveis dinamicamente.

Dúvidas?

