

Um sistema computacional é composto por usuários, hardware e software, sendo os softwares divididos em aplicativos e sistemas operacionais. Os sistemas operacionais (SO) fornecem uma interface entre os aplicativos e o hardware, facilitando o uso e melhorando a eficiência do hardware. Para entender o papel do SO, é importante conhecer sua evolução junto aos sistemas computacionais. Estudaremos os tipos de SO, a estrutura básica do sistema operacional, que inclui o kernel, chamadas de sistema e modos de acesso ao núcleo, além de explorar o uso básico do SO Linux.

Estudar sistemas operacionais é essencial para profissionais de áreas como administração de sistemas, programação de aplicações concorrentes, segurança e redes. Um sistema computacional (S.C.) é composto por:

- **Hardware:** recursos básicos como CPU, memória e dispositivos de E/S.
- **Aplicativos:** programas que utilizam os recursos para resolver problemas, como compiladores e jogos.
- **Usuários:** pessoas, máquinas ou computadores.
- **Sistema operacional:** controla e coordena o uso do hardware entre os programas e usuários.

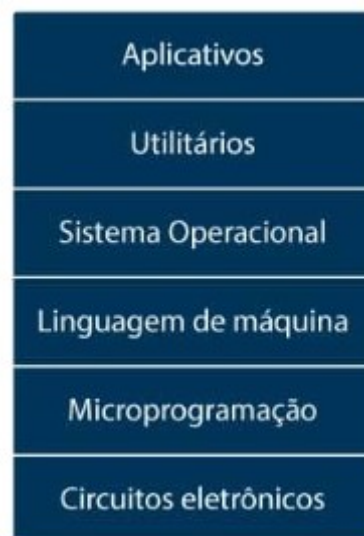
Os sistemas computacionais são complexos e difíceis de gerenciar sem o uso de um sistema operacional, que facilita a interação com o hardware e otimiza o uso dos recursos. Diferente de aplicativos de usuário, o sistema operacional atua como intermediário entre o usuário e o hardware, tornando o uso do computador mais simples, rápido e seguro.

Você provavelmente já usou sistemas operacionais como Windows, Linux, Mac OS X ou Android, mas a interface que interage diretamente com o usuário não faz parte do sistema operacional, embora o utilize. Essa interface pode ser de texto (shell) ou gráfica (GUI).

Os programas em um computador rodam em **dois modos: usuário** (acesso limitado ao hardware) e **kernel** (onde o SO tem acesso completo ao hardware). O sistema operacional, executado no **modo kernel, coordena a execução dos programas e gerencia os recursos de hardware.**

Nos primeiros computadores, a programação era feita por painéis físicos, exigindo amplo conhecimento do hardware, que era pouco acessível ao usuário. O sistema operacional surgiu para tornar o uso do hardware mais eficiente, modularizando e abstraindo a complexidade.

O SO também abstrai o hardware e outras rotinas de software, oferecendo ao usuário uma visão simplificada e acessível, representada em um modelo de **máquina de camadas**. A camada **acima do hardware** é chamada de **máquina virtual**.



O sistema operacional é crucial para gerenciar os recursos do hardware e expandir o conjunto de instruções do computador. Sua evolução é intimamente ligada à arquitetura de computadores e pode ser dividida em várias gerações.

- **Primeira geração (1945-1955)**: Computadores com programação em painéis físicos e uso de válvulas nos circuitos lógicos.
- **Segunda geração (1955-1965)**: Adoção de transistores e sistemas de computação em lote, utilizando leitura de cartões e fitas.
- **Terceira geração (1965-1980)**: Consolidação dos sistemas operacionais, com inovações como multiprocessamento, multiprogramação, time-sharing, e o surgimento do UNIX.
- **Quarta geração (1980-presente)**: Surgimento dos computadores pessoais, microprocessadores, redes distribuídas, novos sistemas operacionais como MS Windows, e avanços como interfaces gráficas, arquiteturas paralelas e processamento distribuído.

Os sistemas operacionais evoluíram para incluir tecnologias como redes locais, sistemas multiusuários, a linguagem C, e o suporte a dispositivos como celulares e tablets, com destaque para os sistemas **Windows** e **Unix**, que continuam influenciando o desenvolvimento tecnológico.

A história do **Windows** começou com o **MS-DOS**, lançado em 1981 como um sistema operacional de 16 bits monoprogramável e monousuário. Em 1985, o **Windows 1.0** foi lançado, trazendo uma interface gráfica para o MS-DOS. As versões subsequentes, como **Windows 3.0**, **Windows 95**, **Windows 98** e **Windows Me**, mantiveram o MS-DOS como núcleo.

Em 1993, a Microsoft lançou o **Windows NT**, um sistema de 32 bits com multitarefa, memória virtual e suporte a múltiplos processadores, com um novo núcleo e compatibilidade parcial com o MS-DOS. O **Windows 2000** foi uma evolução do Windows NT, com novos recursos como plug and play e Active Directory.

A partir do **Windows XP** (2001), a Microsoft integrou as famílias DOS-Windows e Windows NT/2000. Novas versões para desktops (Windows Vista, 7, 8, 10) e servidores (Windows Server 2003, 2008, 2012, 2016, 2019) continuaram sendo lançadas.

O **Unix** tem uma origem diferente. Em 1965, o MIT, Bell Labs e a General Electric criaram o **MULTICS**, um sistema operacional de tempo compartilhado. Em 1969, Ken Thompson desenvolveu uma versão simplificada chamada **Unix** para o minicomputador PDP-7, reescrita posteriormente em linguagem C. O **Unix** foi adotado por várias universidades, incluindo a Universidade de Berkeley, que criou a versão **BSD** com melhorias como memória virtual, C shell e protocolo TCP/IP.

Em 1991, **Linus Torvalds** iniciou o desenvolvimento do **Linux**, baseado no **Minix**, uma versão educacional do Unix. O Linux evoluiu com contribuições de programadores voluntários. Para unificar as versões do Unix, o **comitê POSIX** do IEEE criou um padrão de chamadas e utilitários para sistemas Unix.



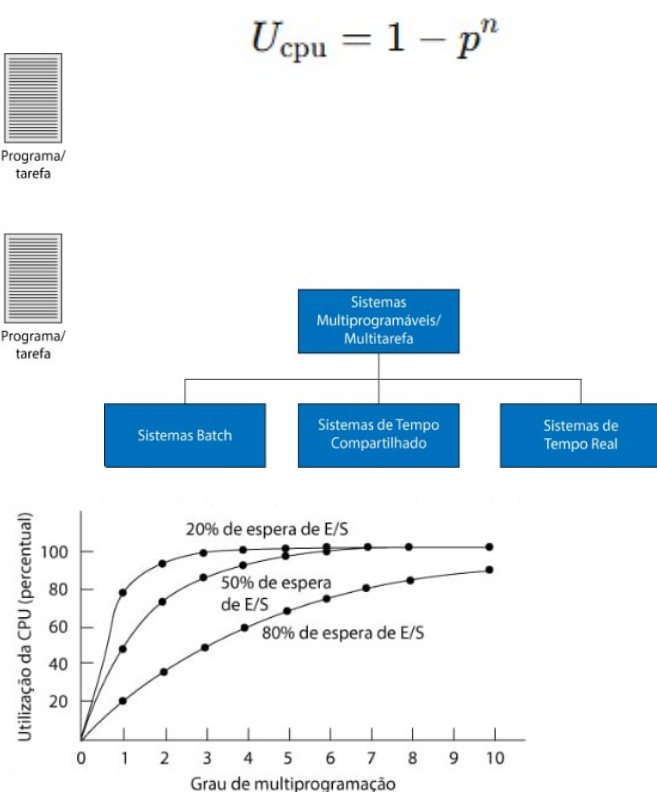
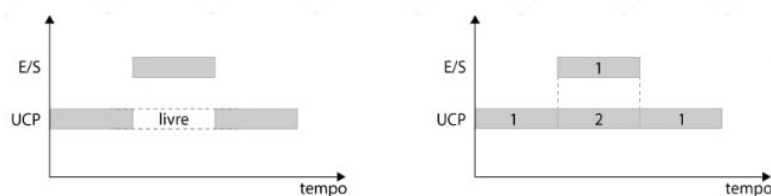
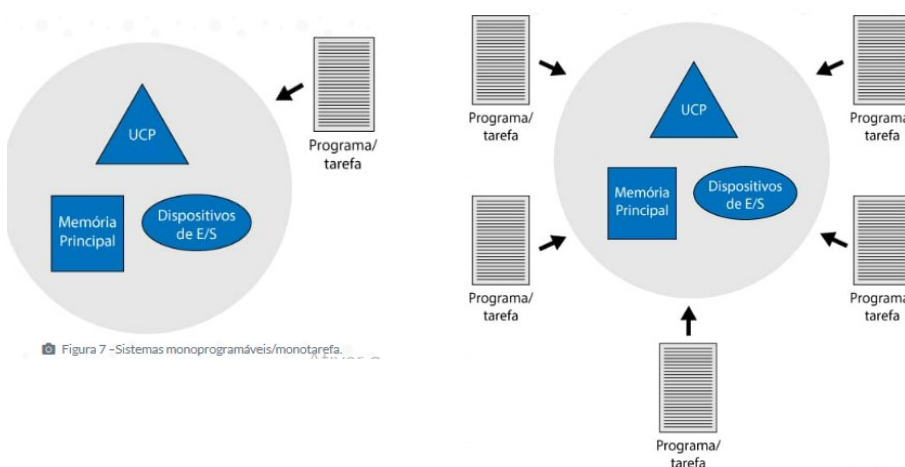
Os sistemas operacionais podem ser classificados em três tipos: **monoprogramáveis/monotarefa**, **multiprogramáveis/multitarefa** e **com múltiplos processadores**.

- **Monoprogramáveis** (ou monotarefa): **Alocam** os **recursos** computacionais (**processador**, **memória** e **periféricos**) exclusivamente **para um único programa**, deixando outras aplicações em espera.
- **Multiprogramáveis** (multitarefa): **Compartilham** os **recursos** entre **várias aplicações**, permitindo a **execução simultânea** de múltiplos programas. Esses sistemas podem ser **batch**, **de tempo compartilhado** ou **de tempo real**. O sistema operacional gerencia vários programas ao mesmo tempo, resolvendo problemas como o desperdício de CPU, comum em sistemas monoprogramáveis, onde o processador fica ocioso enquanto aguarda a conclusão de operações de E/S.
- **Múltiplos processadores**: Utilizam mais de um processador para dividir as tarefas.

**Sistemas batch** (processamento em lote): modelo de execução de tarefas computacionais onde um **conjunto de processos é agrupado e executado sem interação do usuário**. São amplamente utilizados para tarefas que exigem grande processamento de dados, como geração de relatórios, processamento de folha de pagamento e backup de sistemas.

**Sistemas de tempo compartilhado**: modelo de computação onde **múltiplos usuários podem acessar um mesmo sistema simultaneamente**, com a CPU alternando rapidamente entre as tarefas. Esse conceito permite a execução interativa de processos, garantindo que cada usuário tenha a sensação de estar utilizando o sistema sozinho.

A **utilização da CPU** em sistemas multiprogramáveis depende do **número de processos na memória**, chamado de **grau de multiprogramação**. A fórmula para calcular a utilização da CPU é  $U_{cpu} = 1 - p^n$ , onde **p** é o tempo de espera por E/S e **n** o número de processos na memória. Para que a CPU seja mais eficiente, é necessário ter um número suficiente de processos na memória simultaneamente, como demonstrado pela curva de espera de E/S.



Sistemas com **múltiplos processadores** utilizam **dois ou mais processadores trabalhando simultaneamente**, o que oferece **benefícios** como:

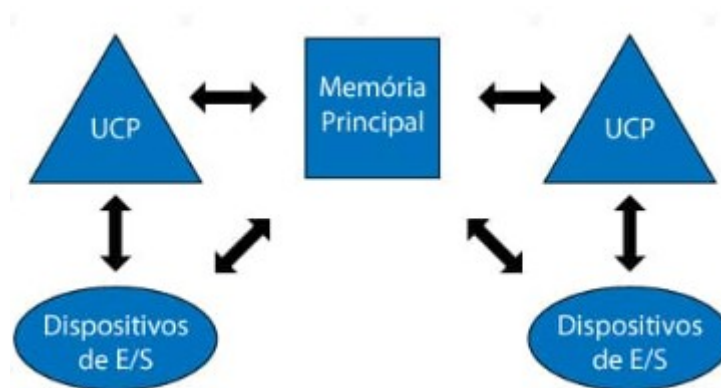
- **Aumento da capacidade de processamento**: A carga de trabalho é distribuída entre os processadores, permitindo um maior desempenho.
- **Redundância**: Se um processador **falhar**, **outro** pode **assumir a carga** de trabalho, garantindo maior confiabilidade.
- **Distribuição de carga**: A carga de trabalho é dividida entre os processadores, otimizando o uso dos recursos computacionais.

Esses sistemas podem ser classificados em:

### Sistemas fortemente acoplados

Nesses sistemas, os **processadores compartilham a memória principal**, e **um único sistema operacional gerencia os recursos**, como periféricos. As características incluem:

- **SMP (Symmetric Multiprocessors)**: **Todos os processadores têm acesso igual à memória**, o que permite um desempenho equilibrado e eficiente ao longo do tempo.
- **NUMA (Non-Uniform Memory Access)**: Existem **conjuntos de processadores e memória**, com o tempo de acesso à memória variando de acordo com a localização física dos processadores. O tempo de acesso é mais rápido para os processadores mais próximos da memória que acessam.



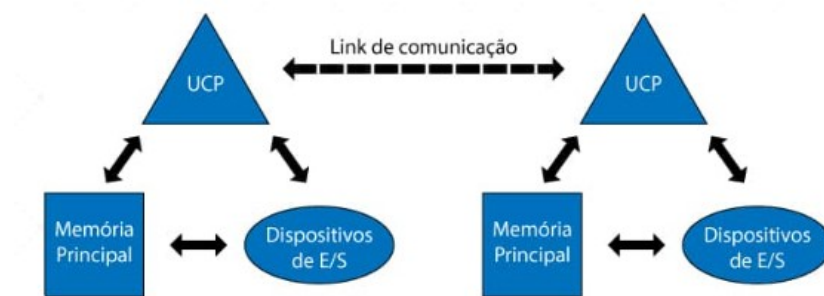
### Sistemas fracamente acoplados

Neste modelo, **dois ou mais sistemas independentes**, **cada um com seu próprio processador, memória e sistema operacional**, estão conectados por uma rede de comunicação. Cada sistema opera de forma autônoma, com seus recursos sendo gerenciados separadamente.

Esses sistemas são conhecidos como "**Sistemas Avançados de Processamento**" e podem incluir:

- **Computadores com múltiplos processadores:** Cada processador tem sua unidade de processamento e memória, mas trabalham de forma colaborativa.
- **Processadores com múltiplos núcleos:** Um único chip contém múltiplos núcleos de processamento que podem executar tarefas simultaneamente.
- **Sistemas distribuídos:** Vários sistemas independentes estão conectados através de uma rede, cada um com seus próprios recursos, e se comunicam para executar tarefas complexas.

O sistema operacional precisa ser adaptado para gerenciar esses sistemas de forma eficiente, coordenando a comunicação e o uso dos recursos de forma otimizada, garantindo a máxima performance.



## Outras Classificações dos Sistemas Operacionais

Ao longo de mais de 50 anos, diversos tipos de sistemas operacionais foram desenvolvidos, alguns amplamente conhecidos, outros menos populares. Cada sistema oferece um conjunto específico de serviços, especialmente em computadores de grande porte, como:

1. **Processamento Simultâneo:** Execução de múltiplas tarefas ao mesmo tempo.
2. **Gerenciamento de E/S:** Suporte ao manuseio de grandes volumes de entrada e saída de dados.
3. **Modos de Processamento:** Processamento em lote (batch), processamento de transações e tempo compartilhado.

## Principais Serviços

- **Sistemas em lote (Batch):** Processam tarefas rotineiras sem interação do usuário, como relatórios de vendas e apólices de seguros.
- **Sistemas de Processamento de Transações:** Administram grandes quantidades de pequenas requisições simultâneas, como **transações bancárias** e **reservas de passagens**.
- **Sistemas de Tempo Compartilhado:** Permitem a execução simultânea de tarefas por múltiplos usuários remotos, como **consultas a bancos de dados**.
- **Sistemas de Tempo Real:** Dependem do tempo como fator crítico, sendo divididos em:
  - **Tempo Real Crítico:** Atrasos podem causar falhas graves, como em **sistemas industriais, aviônicos e militares**.
  - **Tempo Real Não Crítico:** Atrasos não resultam em danos permanentes, como em **sistemas multimídia e telefonia digital**.

## Evolução dos Sistemas Operacionais e Tipos de Licenciamento

Os sistemas operacionais evoluem conforme a tecnologia avança, com alguns tipos e funcionalidades desaparecendo e outros surgindo. Algumas ideias podem se tornar obsoletas e reaparecer no futuro devido a novas demandas tecnológicas.

### Exemplo

O uso de **memória cache** nas CPUs depende da velocidade comparativa entre processadores e memórias. Se as memórias superarem a velocidade das CPUs, a cache se tornará desnecessária. Caso contrário, voltará a ser utilizada.

### Tipos de Licenciamento de Software

- **Software Proprietário:** Licenciado sob copyright, sem acesso ao código-fonte para modificação, restrito à fabricante.
- **Software Livre:** Garante quatro liberdades fundamentais, incluindo **execução, estudo, modificação e redistribuição**. Exemplo: Licença GNU da Free Software Foundation (FSF).
- **Software de Código Aberto:** Código-fonte disponível para modificação, mas sem obrigatoriedade de seguir as quatro liberdades do software livre. Exemplo: Licenças Apache, MIT, Mozilla Public, entre outras.

Embora todo software livre seja de código aberto, nem todo software de código aberto é livre.

## Estrutura do Sistema Operacional

O **sistema operacional** é um conjunto de rotinas que fornecem serviços a usuários, aplicações e ao próprio sistema. O **kernel** (núcleo do SO) gerencia essas funções e está posicionado na estrutura da máquina em níveis.

### Características do Funcionamento

- As **tarefas** são **executadas concorrentemente**, sem ordem fixa.
- Eventos podem ser **assíncronos**, relacionados ao hardware ou ao próprio SO.

### Principais Funções do Kernel

- ✓ Tratamento de interrupções e exceções.
- ✓ Criação, eliminação, sincronização e escalonamento de processos e threads.
- ✓ Gerência de memória, arquivos e dispositivos de E/S.
- ✓ Suporte a redes locais e distribuídas.
- ✓ Controle de segurança, auditoria e contabilização do uso do sistema.

A execução dessas rotinas é controlada pelo mecanismo de **system call**, que **garante segurança** ao **permitir apenas** a **execução de funções autorizadas**.

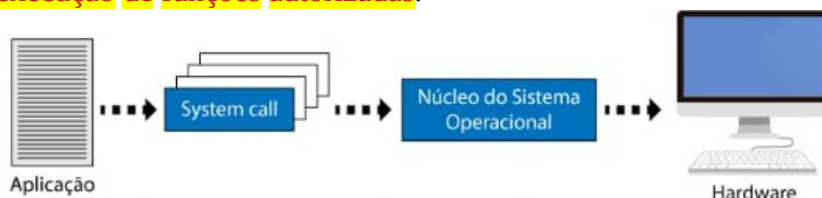


Figura 15 - System call. Fonte: Adaptado de Machado, 2007.

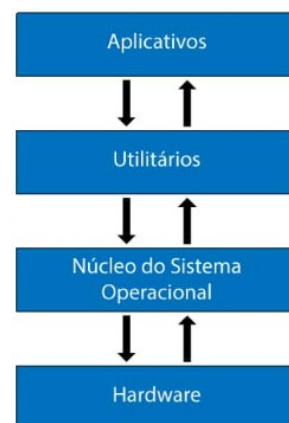


Figura 14 - Kernel do SO na máquina de níveis.



## System Calls

A **system call** é o **mecanismo** que **permite** que uma **aplicação** **acesse rotinas do sistema operacional**. O SO verifica os **privilégios da aplicação** antes de executar a chamada. Caso contrário, bloqueia a operação.

Cada **serviço do SO** tem uma system call específica, e cada sistema operacional possui um conjunto próprio de chamadas, com parâmetros e formas de ativação distintas.

contador = read (arq, buffer, nbytes)

O 1º especifica o arquivo.  
O 2º é um ponteiro para o buffer.  
O 3º dá o número de bytes que deverão ser lidos.

Figura 16 - Chamada ao sistema "read". Fonte: Adaptado de Tanenbaum, 2010.

## Fluxo de Execução de uma System Call

1. O programa armazena os parâmetros na pilha.
2. A rotina de biblioteca é chamada, armazenando o número da chamada de sistema.
3. Uma instrução **TRAP** alterna do **modo usuário** para o **modo núcleo**, iniciando a execução no SO.
4. O kernel verifica o número da chamada e despacha a rotina correspondente.
5. Após a execução, o controle retorna à rotina de biblioteca e, em seguida, ao programa usuário.
6. O programa limpa a pilha e segue sua execução.

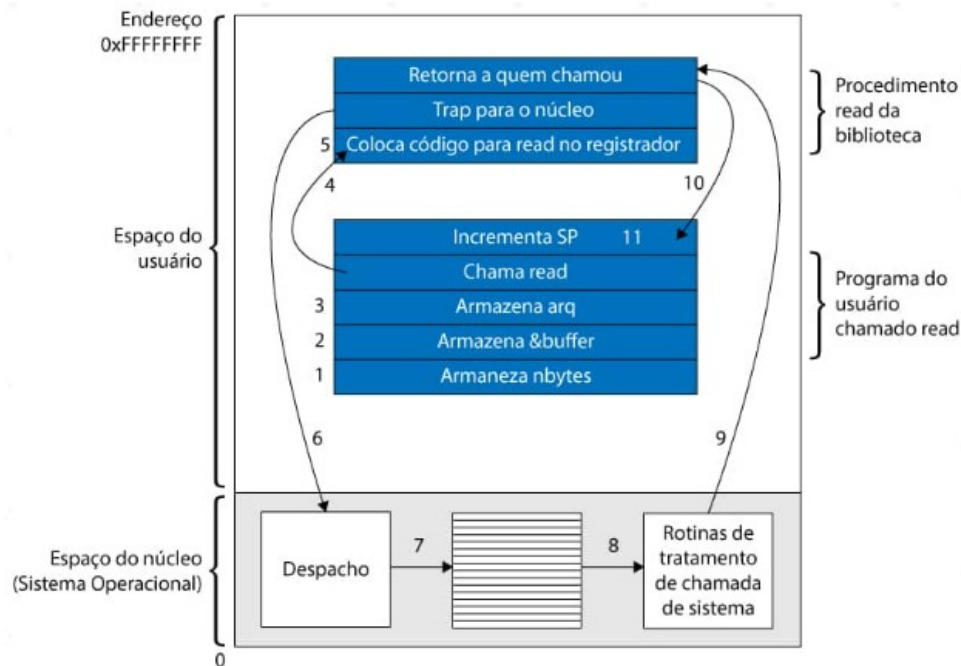


Figura 17 - Passos para a realização da system call "read".

Esse processo garante um acesso seguro e controlado ao núcleo do sistema operacional.

## System Calls e APIs

O **funcionamento de uma system call** envolve três etapas:

1. **Chamada** pela aplicação para executar uma rotina.
2. **Processamento** da solicitação pelo **kernel**.
3. **Retorno** de um estado de conclusão, indicando sucesso ou erro.

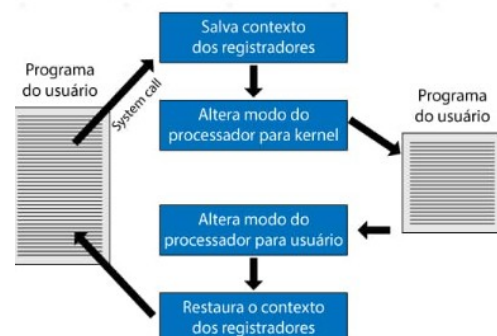


Figura 18 - Chamada a uma rotina de sistema.

## Padrões de System Calls

- **POSIX**: Padrão internacional (**ISO/IEC/IEEE 9945**) para unificar versões do **UNIX**, permitindo compatibilidade entre sistemas.
- **Win32**: Conjunto de chamadas do **Windows**, disponível desde o **Windows 95**. A API **Win32** gerencia a interface gráfica e diferencia chamadas de biblioteca e chamadas reais ao sistema, permitindo compatibilidade entre versões.

## Comparação entre UNIX (POSIX) e Win32

A API **Win32** possui chamadas equivalentes às do **POSIX**, mas com algumas diferenças. Por exemplo, **CreateProcess** combina as funções de **fork** e **execve** do UNIX. Algumas chamadas, como **chmod** e **kill**, não possuem equivalentes diretos no Win32.

## Classificação das System Calls

- **Gerenciamento de Processos**: Criação (**fork()**), espera (**waitpid()**) e encerramento de processos.
- **Gerenciamento de Arquivos**: Abertura (**open()**), leitura (**read()**), escrita (**write()**) e fechamento (**close()**).
- **Gerenciamento de Diretórios**: Criação (**mkdir()**), remoção (**rmdir()**) e montagem de sistemas de arquivos (**mount()**).
- **Outras Funções**: Alteração de permissões (**chmod()**) e obtenção do tempo do sistema (**time()**).

As **system calls** são fundamentais para a comunicação entre aplicações e o sistema operacional, garantindo controle e segurança no acesso aos recursos.

## Modos de Acesso no Sistema Operacional

As **instruções** executadas por aplicações podem ser:

- **Privilegiadas**: Podem comprometer o sistema.
- **Não privilegiadas**: Não oferecem riscos.

Os modos de acesso ao sistema são:

1. **Modo Kernel (Supervisor)**: Aplicações com instruções privilegiadas podem executar todas as instruções e proteger a memória do SO.
2. **Modo Usuário**: Aplicações com instruções não privilegiadas operam com restrições.

Ao chamar uma **System Call**, o SO verifica os privilégios da aplicação. Se não autorizada, impede o acesso via proteção por software. Aplicações devem sempre ser executadas no **modo usuário**.

Caso uma aplicação tente executar uma instrução privilegiada sem autorização, um mecanismo de proteção por **hardware** gerará um erro para garantir a segurança do sistema.

## Arquiteturas de Kernel



As principais arquiteturas de sistemas operacionais são:

1. **Arquitetura Monolítica (Mono-Kernel):** O sistema operacional é composto por vários módulos compilados separadamente e depois unidos, formando um único programa executável. Todos os módulos são visíveis e interagem livremente.
2. **Arquitetura de Camadas:** O sistema é dividido em camadas que oferecem funções para camadas superiores. A principal vantagem é a isolamento e facilidade de manutenção, mas pode comprometer o desempenho devido à comunicação entre camadas. A maioria dos SO comerciais usa duas camadas.

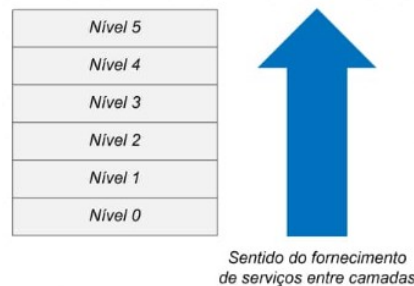


Figura 20 – Arquitetura de camadas.

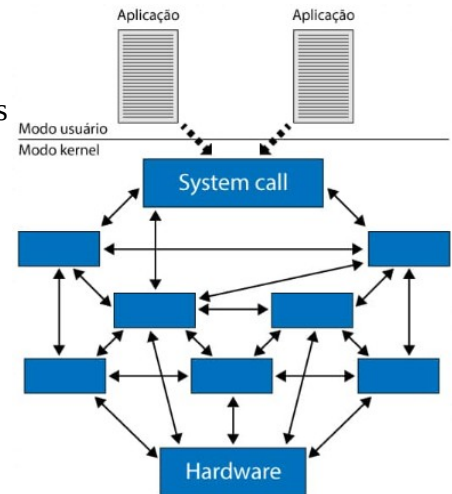


Figura 19 – Arquitetura monolítica.

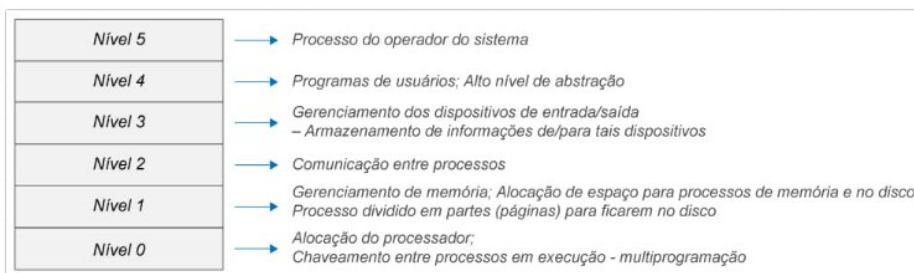


Figura 21 – Arquitetura de camadas.

**Máquina Virtual (VM):** Cria um nível entre o hardware e o sistema operacional, permitindo a virtualização do hardware e criando máquinas virtuais independentes, com seus próprios modos kernel e usuário, interrupções e dispositivos de E/S. Também pode ocultar certas características computacionais por meio de uma camada de abstração.

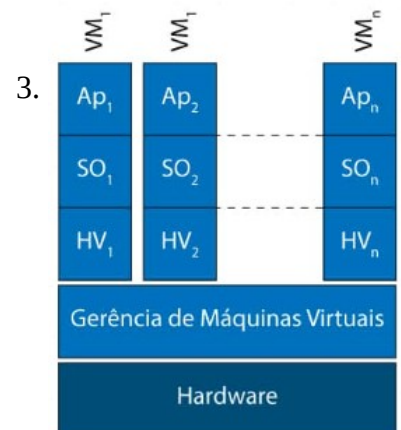


Figura 22 – Máquina virtual.

Nos últimos anos, as máquinas virtuais (VM) têm sido amplamente utilizadas, permitindo que múltiplos sistemas operacionais e softwares sejam executados simultaneamente em uma mesma máquina física, reduzindo custos. Isso só é possível quando a CPU é virtualizável.

**Hypervisors:** O hypervisor é responsável por gerenciar as máquinas virtuais. A mudança de hypervisors de **tipo 1 (executados diretamente no hardware)** para **tipo 2 (executados no sistema operacional hospedeiro)** facilitou a virtualização.

**Arquitetura Microkernel:** Foca em manter o núcleo do sistema operacional pequeno e simples, com serviços essenciais como gerência de arquivos, processos e memória. Módulos adicionais são executados fora do

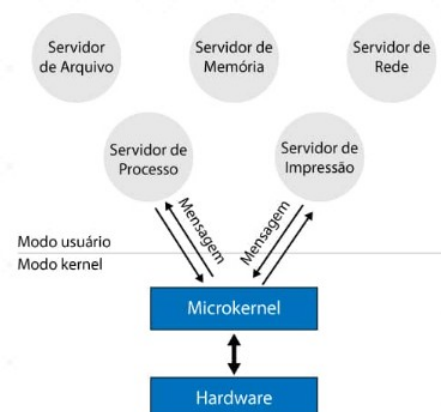


Figura 23 – Arquitetura microkernel.

núcleo. Essa abordagem melhora a confiabilidade, pois falhas em módulos não afetam o sistema inteiro.

**Exonúcleo:** Ao invés de clonar uma máquina virtual, o exonúcleo **divide os recursos entre os usuários, atribuindo blocos específicos do disco a cada máquina virtual.** Isso elimina a necessidade de mapeamento de endereços pelo hypervisor, tornando o sistema mais eficiente.

## Linux

O Linux é um sistema operacional gratuito e moderno, criado por Linus Torvalds em 1991, inicialmente como um kernel pequeno e autocontido, com o objetivo de ser compatível com o UNIX. Ele é considerado "Unix-like" por oferecer recursos de multitarefa e multiusuário.

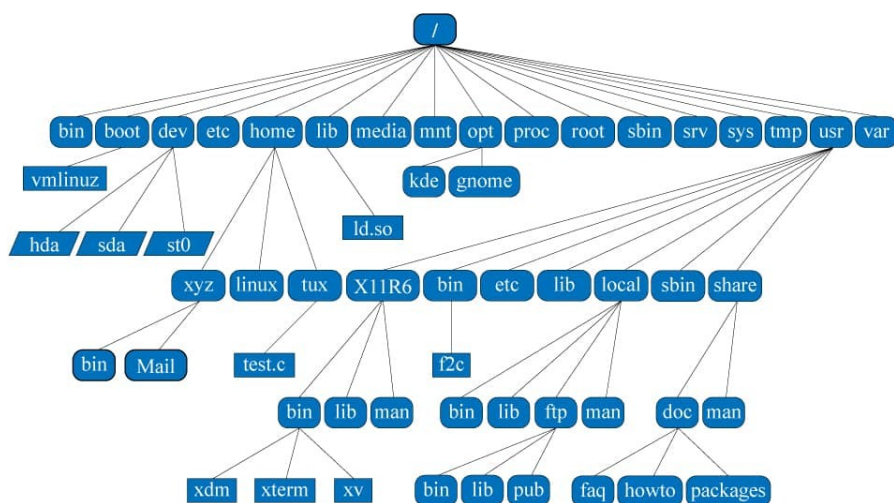
Uma distribuição Linux é composta pelo kernel e um conjunto de aplicativos. As distribuições podem ser instaladas no disco ou executadas de forma "live", através de dispositivos removíveis, carregando o sistema na memória RAM sem a necessidade de instalação.

O Ubuntu, uma distribuição baseada no Debian, será utilizado como exemplo de instalação. Cada versão do Ubuntu recebe um número e um codinome, como o exemplo "20.04 LTS", que garante 5 anos de suporte.

O **Linux** possui uma estrutura de diretórios única, com todos os dispositivos montados abaixo da **raiz "/"**. O sistema é **case-sensitive**, ou seja, diferencia maiúsculas de minúsculas em nomes de arquivos e pastas. O caminho dos diretórios é indicado com "/".

As principais pastas e seus significados são:

- /: Raiz
- /usr: Programas de Usuário
- /bin: Executáveis Binários
- /home: Pasta Pessoal
- /sbin: Sistema Binário
- /boot: Arquivos de Inicialização
- /etc: Arquivos de Configuração
- /lib: Bibliotecas do Sistema
- /dev: Arquivos de Dispositivos
- /opt: Aplicações Opcionais
- /proc: Informação de Processo
- /mnt: Pasta de Montagem
- /var: Arquivos Variáveis
- /media: Dispositivos Removíveis
- /tmp: Arquivos Temporários
- /srv: Serviço de Dados



O Super Usuário, ou "root", tem privilégios administrativos no Linux, assim como o "administrator" no Windows.

Os comandos básicos para o terminal do Linux no Ubuntu são acessados através do ícone "mostrar aplicativos" e digitando "term" na pesquisa. O terminal exibirá uma linha de comando como "teste@teste-VirtualBox:~\$", onde:

- "teste" é o usuário,
- "teste-VirtualBox" é o nome da máquina,
- "~" indica que o usuário está na pasta pessoal (/home/usuário).

Para compilar código-fonte, siga estes passos:

1. Baixe o arquivo-fonte (ex: .tar.gz).
2. Descompacte o arquivo.
3. Acesse o diretório do código-fonte.
4. Execute os comandos:
  - `./configure`
  - `make`
  - `make install`

---

### Instalando o compilador c:

Para instalar o compilador de Linguagem C no Ubuntu, assim como em qualquer distribuição Linux derivada do Debian, basta executar no shell os comandos:

```
sudo apt-get update
sudo apt-get install gcc
```

### Compilando:

Para compilar um programa chamado **prog.c** basta entrar no shell, no diretório onde se encontra o programa, e executar o comando:

```
gcc prog.c
```

Será criado um arquivo executável de nome **a.out**.

### Compilando especificando a saída:

Para compilar um programa e escolher o nome do arquivo executável que será gerado, utilize o parâmetro **-o**. Por exemplo, para compilar o programa **prog.c** e gerar como saída um arquivo executável **prog**, utilize o comando:

```
gcc prog.c -o prog
```

## Executando o programa:

Para executar o programa **prog** que acabou de ser compilado, execute o comando:

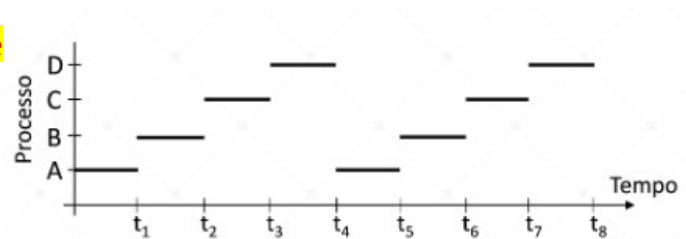
```
./prog
```

Os primeiros sistemas permitiam a execução de apenas um programa por vez, com controle total do sistema. Atualmente, múltiplos programas podem ser carregados e executados simultaneamente, exigindo um controle mais refinado, resultando na noção de **processo**.

Em sistemas multiprogramáveis, a UCP alterna entre processos, dedicando tempo a cada um e criando a ilusão de **pseudoparalelismo**. Todo software no computador é organizado em processos sequenciais, facilitando o tratamento do paralelismo.

Um **processo** é um **programa em execução**, incluindo valores dos registradores, variáveis e espaço de endereçamento. Diferente de um **programa**, que é **passivo**, um **processo** é **ativo**, possuindo um **contador de instruções e registradores**.

**Observação:** Dois processos podem estar associados a um mesmo programa, mas são execuções distintas. Como a UCP alterna entre processos, a velocidade de execução pode variar.



Os processos A, B, C e D se alternam na execução ao longo do tempo. No instante t1, A para e B começa; em t2, ocorre nova troca, e assim sucessivamente até t4, quando D para e A retoma. Como essa alternância ocorre rapidamente, cria-se a ilusão de execução simultânea.

Sistemas operacionais criam processos durante sua operação. Quatro eventos principais levam à criação de processos:

- **Inicialização do sistema:** diversos processos são criados, incluindo processos de **primeiro plano** (interagem com usuários) e **segundo plano** (**daemons**, como servidores de e-mail e impressão).
- **Chamada de sistema de criação de processo:** um processo pode criar outro para auxiliá-lo, útil em multiprocessadores para acelerar tarefas.
- **Solicitação do usuário:** ao executar um programa via comando ou clique, um novo processo é iniciado.
- **Execução de tarefas em lote:** o sistema operacional cria processos conforme há recursos disponíveis para executar as próximas tarefas na fila.

Todos os novos processos são criados por outro já existente por meio de chamadas de sistema. No Linux, a mais comum é **fork()**, que **gera um processo idêntico ao pai**, compartilhando memória, variáveis de ambiente e arquivos abertos.

**Observação:** Todo novo processo é criado por outro já existente por meio de uma chamada de sistema, como **fork() no Linux**, que gera um processo idêntico ao original, compartilhando memória, variáveis de ambiente e arquivos abertos.

O código em C exemplifica a criação de um processo com **fork()**:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int resultado, pid, ppid;
    resultado = fork();
    if (resultado < 0)
        printf("Algo deu errado!!!\n");
    pid = getpid();
    if (resultado == 0) {
        ppid = getppid();
        printf("Eu sou o processo filho, meu PID é %d e meu pai tem PID=%d.\n",
pid, ppid);
    }
    if (resultado > 0) {
        printf("Eu sou o processo pai, meu PID é %d e meu filho tem PID=%d.\n",
pid, resultado);
        waitpid(resultado, NULL, 0);
    }
}
```

**PID: process identifier**

**PPID: parent process identifier**

Quando **fork()** é chamada, a variável **resultado** armazena o PID do filho no processo pai e 0 no filho. A verificação de **resultado** identifica qual processo está executando:

- Se **resultado < 0**, houve erro na criação do processo.
- Se **resultado == 0**, é o processo filho; ele obtém o **PPID** (PID do pai) com **getppid()**.
- Se **resultado > 0**, é o processo pai, que aguarda o término do filho com **waitpid()**.

O **fork()** cria um processo filho idêntico ao pai. Para mudar a execução do filho para outro programa, usa-se **execve()** para alterar sua imagem de memória.

O código a seguir cria três processos filhos, cada um executando um programa diferente usando **execve()**:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
```

```

int main(int argc, char **argv, char* envp[]) {
    int pid, i;

    for (i=1; i<=3; i++) {
        pid = fork();
        if (pid < 0) {
            printf("Algo deu errado!!!\n");
            return 0;
        }
        if (pid == 0) { // Processo filho
            if (i == 1)
                execve("/usr/bin/xcalc", argv, envp);
            if (i == 2)
                execve("/usr/bin/gedit", argv, envp);
            if (i == 3)
                execve("/usr/bin/xeyes", argv, envp);
        }
        else // Processo pai
            waitpid(pid, NULL, 0);
    }
}

```

Cada filho executa um programa: **xcalc**, **gedit** e **xeyes**. O **fork()** cria o processo filho e, no filho, o **execve()** substitui a imagem de memória do processo, carregando o programa desejado.

### Condições de Término de Processo:

- **Saída normal (voluntária):** O processo termina após concluir sua tarefa, sinalizando o sistema operacional.
- **Erro fatal (involuntário):** O processo termina devido a um erro grave.
- **Saída por erro (voluntária):** O processo termina por um erro de programa, como instrução ilegal ou acesso a memória inexistente.
- **Morto por outro processo (involuntário):** O processo é finalizado por outro processo via chamada de sistema.

Em alguns sistemas, a finalização de um processo também encerra todos os processos filhos. No entanto, **Linux** e **Windows** não funcionam dessa forma.

### Resumo:

Em alguns sistemas, processos podem formar uma **hierarquia**, onde o **processo pai** cria **processos filhos**, e esses filhos podem criar outros, gerando uma árvore de processos. No **Linux**, todos os processos, incluindo seus filhos e descendentes, formam um **grupo de processos**. O **processo systemd** (ou **init** dependendo da versão) é o primeiro a ser executado durante a inicialização do sistema e é responsável por iniciar outros processos. Isso cria uma única árvore de processos com o **systemd** (ou **init**) na raiz.

A hierarquia de processos no Linux pode ser visualizada usando o comando **pstree**, que exibe a árvore de processos, como exemplificado abaixo:

```

systemd+-ModemManager---2*[{ModemManager}]
      |-NetworkManager---2*[{NetworkManager}]

```



```

| -VBoxService---8*[{VBoxService}]
| -accounts-daemon---2*[{accounts-daemon}]
| -acpid
| -avahi-daemon---avahi-daemon
| -colord---2*[{colord}]
| -cron
| -cups-browsed---2*[{cups-browsed}]
| -cupsd
| -dbus-daemon
| -gdm3+-gdm-session-wor+-gdm-wayland-ses+-gnome-session-b---
3*[{gnome-session-b}]
|   | | `--2*[{gdm-wayland-ses}]
|   | `--2*[{gdm-session-wor}]
|   `--2*[{gdm3}]
| -2*[{kerneloops}]
| -login---bash---pstree
| -networkd-dispat
| -polkitd---2*[{polkitd}]
| -rsyslogd---3*[{rsyslogd}]
| -rtkit-daemon---2*[{rtkit-daemon}]
| -snapd---8*[{snapd}]
| -switcheroo-cont---2*[{switcheroo-cont}]
| -systemd---(sd-pam)
| -systemd+- (sd-pam)
|   | -at-spi-bus-laun+-dbus-daemon
|   | `--3*[{at-spi-bus-laun}]
|   | -at-spi2-registr---2*[{at-spi2-registr}]
|   | -dbus-daemon
|   | -gjs---4*[{gjs}]
|   | -gnome-keyring-d---3*[{gnome-keyring-d}]
|   | -gnome-session-b---3*[{gnome-session-b}]
|   | -gnome-session-c---{gnome-session-c}
|   | -gnome-shell+-Xwayland
|   |   | -ibus-daemon+-ibus-engine-sim---2*[{ibus-engine-sim}]
|   |   |   | -ibus-memconf---2*[{ibus-memconf}]
|   |   |   `--2*[{ibus-daemon}]
|   |   `--6*[{gnome-shell}]
|   | -goa-daemon---3*[{goa-daemon}]
|   | -goa-identity-se---2*[{goa-identity-se}]
|   | -gsd-a11y-settin---3*[{gsd-a11y-settin}]
|   | -gsd-color---3*[{gsd-color}]
|   | -gsd-keyboard---3*[{gsd-keyboard}]
|   | -gsd-media-keys---3*[{gsd-media-keys}]
|   | -gsd-power---3*[{gsd-power}]
|   | -gsd-print-notif---2*[{gsd-print-notif}]
|   | -gsd-printer---2*[{gsd-printer}]
|   | -gsd-rfkill---2*[{gsd-rfkill}]
|   | -gsd-smartcard---4*[{gsd-smartcard}]
|   | -gsd-sound---3*[{gsd-sound}]
|   | -gsd-usb-protect---3*[{gsd-usb-protect}]
|   | -gsd-wacom---3*[{gsd-wacom}]
|   | -gsd-wwan---3*[{gsd-wwan}]
|   | -gsd-xsettings---3*[{gsd-xsettings}]
|   | -gvfs-afc-volume---3*[{gvfs-afc-volume}]
|   | -gvfs-goa-volume---2*[{gvfs-goa-volume}]
|   | -gvfs-gphoto2-vo---2*[{gvfs-gphoto2-vo}]
|   | -gvfs-mtp-volume---2*[{gvfs-mtp-volume}]
|   | -gvfs-udisks2-vo---3*[{gvfs-udisks2-vo}]
|   | -gvfsd---2*[{gvfsd}]
|   | -gvfsd-fuse---5*[{gvfsd-fuse}]
|   | -ibus-portal---2*[{ibus-portal}]

```

```

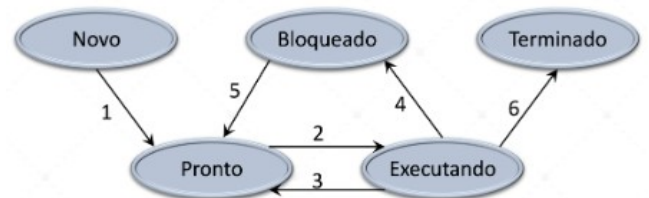
|      |-ibus-x11---2*[{ibus-x11}]
|      |-pulseaudio---3*[{pulseaudio}]
|      |-tracker-miner-f---4*[{tracker-miner-f}]
|      `--xdg-permission----2*[{xdg-permission-}]
|-systemd-journal
|-systemd-logind
|-systemd-resolve
|-systemd-timesyn---{systemd-timesyn}
|-systemd-udev
|-udisksd---4*[{udisksd}]
|-unattended-upgr---{unattended-upgr}
|-upowerd---2*[{upowerd}]
|-whoopsie---2*[{whoopsie}]
`--wpa_supplicant

```

## Resumo Completo:

Um processo em execução pode estar em vários **estados**, que variam conforme o sistema operacional. Os estados mais comuns são: **novo**, **pronto**, **executando**, **bloqueado** e **terminado**.

- **Novo:** O processo foi criado, mas ainda precisa de preparações para iniciar a execução. Quando tudo estiver pronto, o processo transita para o estado **pronto**.
- **Pronto:** O processo está preparado para ser executado, mas aguarda o processador ficar disponível. Quando o processador é alocado, o processo transita para o estado **executando**.
- **Executando:** O processo está sendo executado. Durante esse estado, três situações podem ocorrer:
  1. **Interrupção de tempo:** Se o processo é executado por muito tempo, o sistema operacional pode interrompê-lo e levá-lo de volta ao estado **pronto**, onde aguardará nova chance de execução.
  2. **Solicitação de I/O:** Se o processo solicitar uma operação de entrada/saída, ele vai para o estado **bloqueado** até a operação ser concluída.
  3. **Término:** Se o processo completar sua execução, ele transita para o estado **terminado**.



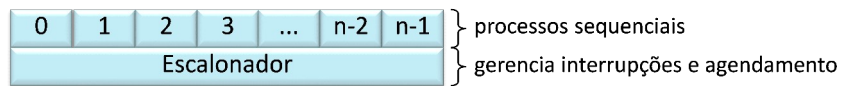
Essas transições entre estados permitem que o sistema operacional gerencie a execução de múltiplos processos de forma eficiente.

Quando um processo entra no **estado bloqueado**, ele permanece lá até a conclusão da operação que estava aguardando. Após isso, o processo transita para o **estado pronto** até ser selecionado novamente para execução.

O **estado terminado** é para processos que não serão mais executados. O sistema operacional desaloca os recursos usados pelo processo antes de removê-lo do sistema.

No Linux, a **tabela de processos** armazena as informações de cada processo no **Bloco de Controle de Processo (BCP)**, que inclui:

- Estado do processo
- Prioridade
- Número do processo
- Registradores da UCP
- Informações de memória
- Contabilidade
- Operações de E/S



O **escalonador** gerencia a execução dos processos, realizando a troca de processos a cada 100ms. Ele é responsável por selecionar quais processos serão executados e deve ser eficiente, pois é chamado com frequência.

A **mudança de contexto** é o processo de transferir o controle da UCP de um processo para outro, salvando o estado do processo em execução e carregando o do processo a ser executado. Esse processo pode durar de 1 a 1000 microssegundos, dependendo de fatores como velocidade da memória e registradores.

O **contexto de um processo** pode ser dividido em três elementos:

1. **Contexto de hardware:** Contém o conteúdo dos registradores. É salvo quando o processo perde a UCP e é essencial para sistemas multiprogramáveis.
2. **Contexto de software:** Define informações como identificação (PID, UID, GID), quotas (limites de recursos) e privilégios (ações permitidas no sistema).
3. **Espaço de endereçamento:** Área de memória onde o programa e seus dados são armazenados. Cada processo tem seu próprio espaço, protegido dos demais.

No Linux, os processos funcionam de maneira sequencial e multitarefa, permitindo a execução simultânea de processos de diferentes usuários. Mesmo com um único usuário logado, diversos **daemons** (processos em segundo plano) podem estar em execução, como o **daemon de impressão**, que gerencia o envio de trabalhos para a impressora.

Os processos são criados usando a chamada de sistema **fork()**, que gera um processo filho com uma cópia do espaço de endereçamento do pai. Arquivos abertos antes de **fork()** permanecem compartilhados entre os dois processos.

O Linux oferece chamadas de sistema para gerenciar processos, como:

- **fork():** Cria um processo filho.
- **waitpid():** Espera a conclusão de um processo filho.
- **execve():** Substitui o processo atual por outro.
- **exit():** Termina a execução de um processo e retorna um status.

## Subprocesso

Um subprocesso (ou processo filho) é criado por um processo pai e pode, por sua vez, gerar outros subprocessos. Essa estratégia permite dividir uma aplicação em partes que podem trabalhar concorrentemente. Por exemplo, em um servidor web, subprocessos podem atender requisições simultâneas, aproveitando o multiprocessamento e evitando que uma requisição simples fique atrasada devido a requisições complexas.

## Utilização de Subprocessos

A criação de subprocessos permite que um servidor web aproveite a capacidade de multiprocessamento, processando requisições simultaneamente. Isso evita que requisições simples fiquem presas atrás de tarefas complexas, garantindo respostas mais rápidas.

Em vez de um único processo atender todas as requisições sequencialmente, o servidor cria um novo subprocesso para cada solicitação. Dessa forma, as requisições são distribuídas entre os processadores do sistema, otimizando o desempenho.

O código em C exemplifica essa abordagem:

1. **pega\_proxima\_requisicao()** – Obtém a próxima requisição. Se não houver nenhuma, o processo pai aguarda sem impactar o desempenho.
2. **fork()** – Cria um subprocesso para processar a requisição.
3. **processa\_requisicao()** – Executado apenas pelo subprocesso, processa a requisição recebida.
4. **exit(0)** – Encerra o subprocesso após concluir o processamento.
5. **while(1) { ... }** – Mantém o servidor sempre pronto para novas requisições.

```
while (1) { // Loop infinito
    req = pega_proxima_requisicao();
    pid = fork();
    if (pid == 0) { // Processo filho
        processa_requisicao(req);
        exit(0);
    }
}
```

Apesar das vantagens, a criação de subprocessos consome recursos, pois exige alocação de contexto de hardware, software e espaço de endereçamento, além de tempo de UCP para gerenciamento dos processos e seus respectivos BCPs (Blocos de Controle de Processos).

## Threads

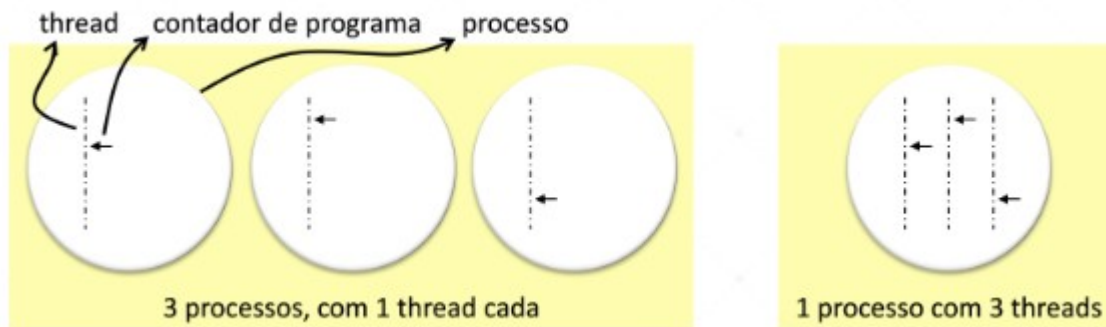
Threads reduzem o tempo e os recursos gastos na criação e eliminação de processos, permitindo a execução concorrente dentro de um mesmo processo sem a necessidade de múltiplos processos.

Cada thread possui seu próprio conjunto de registradores (contexto de hardware), mas compartilha o espaço de endereçamento com os demais threads do mesmo processo. Assim, ao perder o processador, suas informações são salvas, e ele passa pelos mesmos estados que um processo.

A principal diferença entre threads e subprocessos está no espaço de endereçamento:

- **Subprocessos** possuem espaços independentes e protegidos.
- **Threads** compartilham o mesmo espaço de endereçamento, permitindo acesso direto aos dados uns dos outros, sem proteção, e são projetados para trabalhar cooperativamente.

A mudança de contexto entre threads é mais eficiente que entre processos, pois exige apenas a alteração dos registradores, sem necessidade de gerenciamento de memória adicional.



## Threads em um Mesmo Processo

Quando um processo contém múltiplos threads, alguns campos da tabela de processos são específicos de cada thread.

Threads podem ser gerenciados no espaço do usuário, sem envolvimento do sistema operacional, como no pacote **P-threads (POSIX)**. Isso torna a comutação mais rápida, pois não requer chamadas ao kernel. No entanto, se um thread bloquear, todo o processo também será bloqueado, além de afetar a distribuição de tempo entre threads de diferentes processos.

Threads são especialmente úteis em **sistemas com múltiplas UCPs**, onde o paralelismo real é possível, permitindo que diferentes partes de um processo sejam executadas simultaneamente em processadores distintos. Algumas UCPs oferecem suporte direto para **multithread**, possibilitando chaveamento de threads em nanossegundos.

Um processo com um único thread é chamado de **monothread**. Threads podem compartilhar o espaço de endereçamento, arquivos abertos, processos filhos, alarmes e sinais. No entanto, **cada thread tem sua própria pilha**, onde são armazenadas as variáveis locais e os endereços de retorno de cada chamada de rotina.

## Criação de Threads no Linux

No Linux, threads são criados com a chamada de sistema **clone()**, cuja sintaxe é:

```
int clone(int (*fn)(void *), void *stack, int flags, void *arg);
```

Os **principais flags** e seus comportamentos:

- **CLONE\_VM**: Cria um thread (se ausente, cria um processo).
- **CLONE\_FS**: Compartilha informações do sistema de arquivos (se ausente, não compartilha).
- **CLONE\_FILES**: Compartilha descritores de arquivos (se ausente, copia).
- **CLONE\_SIGHAND**: Compartilha a tabela do tratador de sinais (se ausente, copia).
- **CLONE\_PARENT**: O novo thread mantém o mesmo pai do chamador (se ausente, o chamador é o pai).
- **SIGCHLD**: O thread envia **SIGCHLD** ao pai ao terminar (se ausente, não envia).

Um exemplo em **Linguagem C** ilustra a criação de **três threads dentro de um processo**, onde o thread principal finaliza **após a execução do último thread**.

```

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/wait.h>

#define TAMANHO_PILHA 65536
#define _1SEGUNDO 1000000

int global = 0; // Variável global alterada pelos threads

static int funcaoThread(void *arg) {
int id = *((int *) arg); // Identificação de cada thread
int i;

printf("Iniciou thread [%d]\n", id);
for (i=0; i<3; i++) { // Loop no qual o thread altera a variável
global
    printf("Thread [%d] incrementou \"global\" para %d.\n", id, +
+global);
    usleep(_1SEGUNDO * (1+id/10.0));
}
printf("Saindo do thread [%d]\n", id);
}

int main () {
void *pilha;
int i, pid[3];
int id[3] = {1,2,3}; // Identificação a ser passada para cada
thread

for (i=0; i<3; i++) { // Alocando espaço para a pilha de cada
thread
if ((pilha = malloc(TAMANHO_PILHA)) == 0) {
perror("Erro na alocação da pilha.");
exit(1);
}
pid[i] = clone(funcaoThread,
pilha + TAMANHO_PILHA,
CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD,
&(id[i])); // Criação de cada thread
}

```



```

printf("Thread principal aguardando demais threads terminarem.\n");
for (i=0; i < 3; i++)
if (waitpid(pid[i], 0, 0) == -1) { // Aguarda até o término do thread
perror("waitpid")
exit(2);
}
printf("Thread principal terminando.\n\n");
}

```

Quando executado, o programa fornece a seguinte saída:

```

Thread principal aguardando demais threads terminarem.
Iniciou thread [3]
Thread [3] incrementou "global" para 1.
Iniciou thread [2]
Thread [2] incrementou "global" para 2.
Iniciou thread [1]
Thread [1] incrementou "global" para 3.
Thread [1] incrementou "global" para 4.
Thread [2] incrementou "global" para 5.
Thread [3] incrementou "global" para 6.
Thread [1] incrementou "global" para 7.
Thread [2] incrementou "global" para 8.
Thread [3] incrementou "global" para 9.
Saindo do thread [1]
Saindo do thread [2]
Saindo do thread [3]
Thread principal terminando.

```

Pela execução, você pode perceber que os valores das variáveis locais de cada thread não são influenciados pela execução dos demais threads. Porém, quando a variável “global” é modificada por um thread, esta alteração é vista imediatamente por todos os demais threads.

## Tipos de Processos

Os processos podem ser classificados em dois tipos:

- **CPU-bound:** Passam a maior parte do tempo executando, com poucas operações de E/S. São comuns em aplicações científicas.
- **I/O-bound:** Ficam frequentemente no estado bloqueado devido ao alto número de operações de E/S. São comuns em aplicações comerciais e interativas.

## Processos e Threads no Linux

O Linux pode duplicar processos com `fork()` e criar threads com `clone()`, mas trata ambos como tarefas (*tasks*), sem distinção entre processo e thread. Um processo com um único thread é uma única tarefa, enquanto um processo com  $n$  threads terá  $n$  estruturas de tarefas.

A chamada `clone()`, sem flags, se comporta como `fork()`. O contexto de uma tarefa não fica em uma única estrutura, mas em várias, acessadas por ponteiros, facilitando o compartilhamento de informações entre tarefas.

Para compatibilidade com UNIX, cada thread recebe um PID diferente. Inicialmente, processos pai e filho compartilham a mesma memória, e a cópia ocorre apenas quando um deles tenta escrever nela (*copy on write*). Esse mecanismo melhora o desempenho e reduz o consumo de memória.

## Processos de Aplicações Concorrentes

Os sistemas multiprogramáveis possibilitaram a execução concorrente de diferentes partes de um programa, criando as chamadas **aplicações concorrentes**. Nessas aplicações, processos frequentemente compartilham recursos, o que pode gerar problemas que comprometem o sistema.

Por exemplo, em um buffer compartilhado, um processo só pode gravar se houver espaço, e outro só pode ler se houver dados. Isso exige que os processos aguardem até que o buffer esteja pronto para uso.

Isso levanta três questões fundamentais:

1. Como um processo passa informações para outro?
2. Como garantir que não interfiram entre si?
3. Como coordenar processos dependentes?

A comunicação entre processos pode ocorrer por **memória compartilhada** ou **troca de mensagens**, mas, independentemente do método, a sincronização é essencial. Os mecanismos que garantem essa comunicação e o controle do acesso a recursos compartilhados são chamados **mecanismos de sincronização** e também se aplicam a threads.

## Condição de Corrida

Em sistemas onde processos compartilham memória, como ao ler e escrever dados, pode ocorrer **condição de corrida**. Isso acontece quando múltiplos processos acessam dados simultaneamente, e o resultado depende da ordem de execução.

### Exemplo:

Um programa atualiza o saldo de uma conta bancária. Se dois caixas, Carlos e Orlando, tentam atualizar o saldo de 500 simultaneamente, Carlos pode ler o saldo antes de Orlando, adicionar 100 e gravar 600, enquanto Orlando, com o saldo ainda em 500, subtrai 200 e grava 300. O saldo final será 300, e a operação de Carlos (depósito) é perdida, gerando inconsistência.

A condição de corrida ocorre quando o processamento compartilhado resulta em um valor incorreto devido à ordem de execução dos processos.

```
void atualiza_saldo(double valor, int conta) {  
    Registro registro;  
    registro = le_registro(conta);  
    registro.saldo = registro.saldo + valor;  
    grava_registro(registro, conta);  
}
```

## Região Crítica

Para evitar condições de corrida, deve-se garantir que apenas um processo acesse um recurso compartilhado de cada vez, implementando **exclusão mútua**. A parte do programa que acessa a memória compartilhada é chamada de **região crítica (RC)**. Durante a execução na região crítica, outros processos devem aguardar sua vez.

### Requisitos para uma solução eficaz:

1. Apenas um processo pode estar na região crítica por vez.
2. Não deve haver suposições sobre a velocidade ou número de UCPs.
3. Processos fora da região crítica não podem bloquear outros.
4. Nenhum processo deve esperar indefinidamente (evitar **starvation**).

Para garantir a exclusão mútua, os processos devem acessar os recursos compartilhados de maneira sincronizada.

## Semáforos

Um semáforo é uma variável inteira que mantém uma contagem de sinais e utiliza duas operações fundamentais: **down** e **up**.

- **down:**
  - Se o valor do semáforo for maior que 0, ele é decrementado, permitindo que o processo acesse a região crítica.
  - Se o valor for 0, o processo fica bloqueado numa fila de espera até que o recurso seja liberado.
- **up:**
  - Se não houver processos bloqueados, incrementa o valor do semáforo.
  - Se houver processos na fila, desbloqueia um deles em vez de apenas incrementar.

Essas operações são executadas de forma atômica pelo sistema operacional, garantindo a **exclusão mútua**. Assim, ao acessar um recurso compartilhado, um processo realiza **down(mutex)** para entrar na região crítica e **up(mutex)** ao sair, prevenindo que outros processos acessem o recurso simultaneamente.

Por exemplo, no código de atualização de saldo de uma conta, o uso de **down(mutex)** impede que dois processos leiam e modifiquem o mesmo saldo ao mesmo tempo. No cenário dos caixas Carlos e Orlando, o semáforo garante que:

- Carlos, ao acessar a conta, bloqueia o acesso por meio de **down(mutex)**, atualiza o saldo, e então libera o recurso com **up(mutex)**.
- Orlando, ao tentar acessar a conta enquanto Carlos está na região crítica, fica bloqueado até que o semáforo seja liberado, garantindo que ele leia o saldo atualizado.

```
void atualiza_saldo(double valor, int conta) {  
    Registro registro;  
  
    down(mutex);  
  
    registro = le_registro(conta);  
  
    registro.saldo = registro.saldo + valor;  
  
    grava_registro(registro, conta);  
  
    up(mutex);  
}
```

Dessa forma, o uso de semáforos assegura que apenas um processo esteja na região crítica a qualquer momento, eliminando condições de corrida e inconsistências no acesso a recursos compartilhados.

## Monitores

Monitores são mecanismos de sincronização de alto nível que facilitam o desenvolvimento e a correção de programas concorrentes, evitando os problemas complexos e imprevisíveis associados ao uso de semáforos. Um monitor é um pacote que agrupa variáveis, procedimentos e estruturas de dados, permitindo que apenas um processo execute um procedimento de monitor de cada vez. Ao chamar um procedimento do monitor, o processo verifica se outro já está em execução; se sim, ele aguarda sua vez. As variáveis globais do monitor são visíveis apenas dentro do monitor e seus procedimentos, e o bloco de comandos do monitor, que inicializa essas variáveis, é executado apenas uma vez. Cabe ao compilador implementar a exclusão mútua na entrada do monitor, como exemplificado pela cláusula **synchronized** da linguagem Java, que transforma regiões críticas em procedimentos de monitor para simplificar o desenvolvimento concorrente.



## Sincronização no Linux

O Linux oferece semáforos para sincronização de tarefas, utilizando as chamadas de sistema **sem\_wait()** (realiza a operação up()) e **sem\_post()** (realiza a operação down()). Quando **sem\_wait()** é chamada e o semáforo é zero, a tarefa é bloqueada até ser liberada por **sem\_post()**. Para verificar sem bloquear, pode-se usar **sem\_trywait()**, que retorna erro sem bloqueio caso não seja possível executar a operação.

Além de semáforos, o Linux permite comunicação entre processos por troca de mensagens e sinais. A comunicação via mensagens pode ser feita através de **pipes**, um mecanismo que conecta a saída de um processo à entrada de outro. Por exemplo, `*cat .txt | sort | uniq` combina arquivos .txt, ordena as linhas e remove duplicatas.

## Comandos e Comunicação no Linux

- `*cat .txt` exibe o conteúdo dos arquivos .txt do diretório na saída padrão (tela).
- `sort` ordena as linhas recebidas e as envia para a saída padrão.
- `uniq` elimina linhas duplicadas e envia as únicas para a saída padrão.

A **barra vertical (|)** entre comandos cria um **pipe**, conectando a saída de um comando à entrada de outro. No exemplo, a saída de `*cat .txt` vai para **sort**, que ordena, e a saída deste vai para **uniq**, que remove duplicatas.

A comunicação entre processos no Linux também pode ser feita por **sinais**, enviados para tratar eventos, como o fim de um temporizador ou o término de um processo filho. Para isso, é necessária uma rotina de tratamento.

## Sinais no Linux

- **Sinais e suas ações padrão:**
  - **SIGHUP:** Terminar, gerado pelo fim do terminal controlador.
  - **SIGTERM:** Terminar, informa que o processo deve parar.
  - **SIGINT:** Terminar, interrupção do terminal controlador.
  - **SIGKILL:** Forçar término do processo (não pode ser ignorado).
  - **SIGTSTP:** Suspende, suspende o processo (pode ser retomado).
  - **SIGSTOP:** Suspende, similar ao SIGTSTP, mas não pode ser sobrescrito.
  - **SIGCONT:** Retornar à execução se estiver suspenso.
  - **SIGCHLD:** Ignorar, informa ao processo pai sobre o término ou suspensão de um filho.
  - **SIGALRM:** Terminar, fim de temporizador.
  - **SIGURG:** Ignorar, condição urgente no socket (geralmente, chegada de pacote de rede).
  - **SIGUSR1/SIGUSR2:** Terminar, sinais definidos pelo usuário.
- **Ações ao receber um sinal:**
  - Tratamento padrão (definido pelo kernel).
  - Captura, para tratamento por função do usuário.
  - Ignorar.
- **Sinais que não podem ser ignorados ou capturados:**
  - **SIGKILL** e **SIGSTOP** sempre executam o tratamento padrão.
- **Fontes de sinais:**
  - Exceções de hardware, condições de software, comandos **kill**, chamadas de sistema, e combinações de teclas como **Ctrl + C**.

Chamada de sistema	Descrição
signal()	Instala rotina para tratamento do sinal.
sigaction()	Define a ação a ser tomada nos sinais.
sigreturn()	Retorna de um sinal.
sigpending()	Obtém o conjunto de sinais bloqueados.
kill ()	Envia um sinal para um processo.
alarm()	Ajusta o alarme do relógio para envio de um sinal.
pause()	Suspende o chamador até o próximo sinal.

## Escalonamento

A multiprogramação permite que vários

processos compartilhem a UCP (Unidade Central de Processamento), maximizando sua utilização. O escalonamento é o processo que define a ordem de execução dos processos que competem pela UCP, sendo realizado pelo escalonador do sistema operacional. Sempre que a UCP está ociosa, o escalonador escolhe um processo pronto na memória para execução.

O algoritmo de escalonamento pode ser **preemptivo** (interrompe processos em execução) ou **não preemptivo** (processo continua até ser concluído). O escalonamento preemptivo exige interrupção de relógio para transferir o controle da UCP ao escalonador, gerando overhead devido à troca de contexto. Já o escalonamento não preemptivo evita condições de corrida, pois um processo não pode ser interrompido durante sua execução.

### Resumo das diferenças principais:

- **Preemptivo:** Pode interromper processos em execução.
- **Não preemptivo:** Não permite interrupção de processos enquanto executando.

### Tipos de Escalonamento

1. **FIFO (First In, First Out):** O primeiro processo a entrar é o primeiro a ser executado. É fácil de programar, mas pode levar a longos tempos de espera para processos curtos e prejudicar processos I/O-bound. Não preemptivo.
2. **SJF (Shortest Job First):** Seleciona o processo com menor tempo de execução. Reduz o tempo médio de espera em relação ao FIFO, mas é difícil prever o tempo de execução de um processo. Não preemptivo.
3. **SRTN (Shortest Remaining Time Next):** Versão preemptiva do SJF. O processo com o menor tempo restante é escolhido. Permite melhor desempenho para processos curtos.
4. **Escalonamento Cooperativo:** O processo libera voluntariamente a UCP após um tempo de execução. Pode gerar problemas se um processo não liberar a UCP. Não preemptivo.
5. **Round Robin:** Similar ao FIFO, mas cada processo tem um tempo limitado (quantum) para usar a UCP. Preemptivo. Se o quantum for pequeno, há overhead; se for grande, a interatividade é afetada.
6. **Escalonamento por Prioridade:** Processos têm prioridades associadas. O sistema escala preferencialmente os processos com maior prioridade. Pode ser preemptivo com ajuste de prioridades durante a execução.
7. **Escalonamento por Múltiplas Filas:** Processos são classificados em diferentes filas com características de processamento distintas. Cada fila tem seu próprio mecanismo de escalonamento e prioridade.
8. **Escalonamento em Sistemas de Tempo Real:** Necessário para sistemas que precisam reagir a estímulos externos dentro de prazos fixos. Pode ser crítico ou não crítico. Algoritmos podem ser estáticos (decisões antes de execução) ou dinâmicos (decisões durante a execução).
9. **Escalonamento de Threads:** Quando múltiplos processos têm threads, o escalonamento pode ser por threads de usuário ou de núcleo. Threads de usuário são controladas pelo processo, enquanto threads de núcleo são controladas pelo sistema operacional, com maior overhead.



## Escalonamento no Linux

O Linux utiliza multitarefa preemptiva, onde o escalonador decide quais processos executar e quando. Ele é baseado em três classes de tarefas: FIFO em tempo real, escalonamento circular em tempo real e tempo compartilhado.

1. **FIFO em tempo real:** Escalonado por prioridade, sem preempção. Só é interrompido por tarefas FIFO de maior prioridade.
2. **Escalonamento circular em tempo real (Round Robin):** Tarefas têm um quantum. Excedido o quantum, ocorre preempção, e a tarefa vai para o final da fila.
3. **Tempo compartilhado:** Escalonado apenas se não houver tarefas de tempo real. As prioridades variam de 100 a 139.

Embora chamadas de "tempo real", essas classes não garantem limites de tempo de sistemas reais de tempo real.

O Linux tem 140 níveis de prioridade, sendo 0 a 99 para tarefas de tempo real e 100 a 139 para tempo compartilhado.

O comando **nice** altera a prioridade das tarefas, permitindo apenas a redução de prioridade por usuários comuns, enquanto o root pode aumentar ou diminuir a prioridade.

## Gerência de Memória

A memória de um computador é composta por diferentes tipos de armazenamento, como registradores, cache (armazenamento interno), memória RAM (primária), e armazenamento secundário (discos SSD, magnéticos, óticos e fitas magnéticas). O gerenciamento eficiente da memória é crucial, considerando as diferenças de custo, velocidade e capacidade de cada tipo de memória.

Existem três grandes tipos de armazenamento:

1. **Armazenamento Interno:** Composto pelos registradores e cache, é a memória de trabalho do processador e volátil.
2. **Armazenamento Primário:** Refere-se à memória RAM, usada para armazenar programas e dados temporários. É volátil e acessada diretamente pela CPU.
3. **Armazenamento Secundário:** Também chamado de armazenamento de massa, é não volátil e armazena dados permanentemente. Para ser usado, deve ser transferido para a memória primária.

O sistema operacional gerencia a memória, realizando funções como alocar e desalocar processos, controlar o espaço disponível, garantir que os processos não acessem a memória de outros e transferir dados entre a memória primária e o armazenamento secundário.

## Como os processos enxergam a memória

A memória de um computador é composta por circuitos eletrônicos que armazenam bits, sendo chamada de memória física, com endereços representados em hexadecimal. Do ponto de vista do processo, a memória é vista como um conjunto de endereços lógicos, acessados pelos comandos da linguagem de máquina. Essa memória lógica é frequentemente maior que a memória física, gerando a memória virtual.

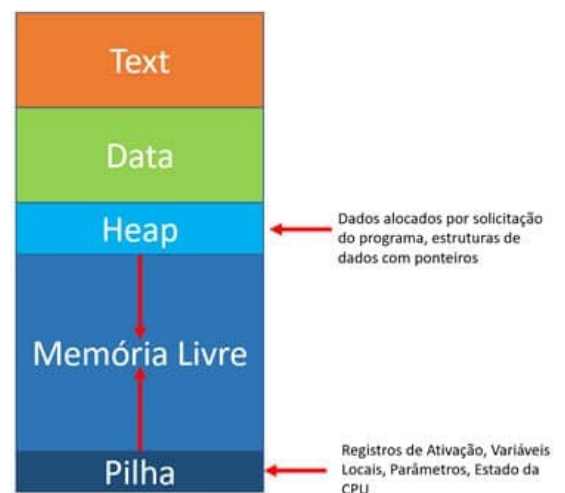
Os processos não acessam a memória física diretamente, mas sim a memória lógica, com um espaço de endereçamento único e independente para cada processo.

### Modelo de memória de um processo

O modelo de memória de um processo é composto por:

- **Text:** Armazena o código do programa.
- **Data:** Armazena as variáveis do programa.
- **Heap:** Aloca variáveis temporárias e dinâmicas.
- **Pilha:** Armazena registros de ativação e variáveis locais.

Entre o **Heap** e a **Pilha**, há uma área livre que permite que eles cresçam em direção um ao outro. Como o processo usa endereços lógicos e o hardware usa endereços físicos, é necessário traduzir o endereço lógico para o físico para o correto mapeamento.



### Mapeamento de memória e MMU

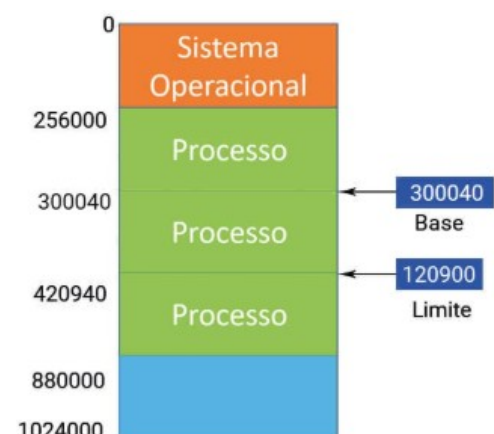
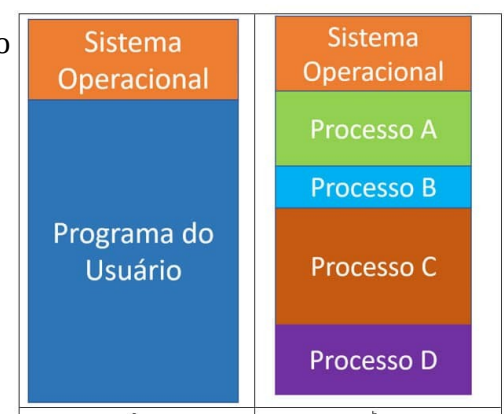
A **Memory Management Unit (MMU)** realiza o mapeamento de endereços lógicos para físicos com base nas seguintes regras:

- Se o endereço lógico está dentro dos limites, é válido e igual ao endereço físico.
- Se o endereço lógico está abaixo do limite superior, soma-se o endereço base e acessa-se o endereço resultante (endereço lógico diferente do físico).
- Se o endereço lógico está fora dos limites, é inválido e gera exceção.

A memória de um processo é exclusiva, mas compartilhada entre processos, tanto em sistemas monotarefa (com área dedicada ao SO) quanto multiprogramados. É necessário evitar que um processo acesse a memória de outro para prevenir erros.

### Proteção de Memória

A proteção de memória assegura que cada processo tenha um espaço exclusivo na memória, permitindo execução concorrente sem

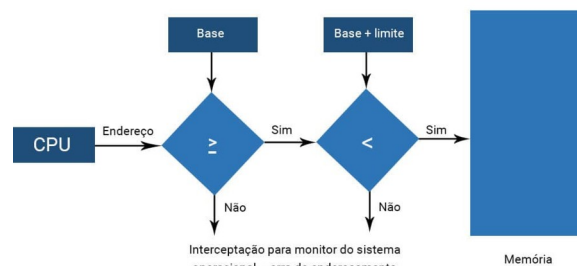


interferência. Para isso, define-se um intervalo de endereços legais que um processo pode acessar, garantindo que o acesso seja restrito ao seu próprio espaço.

A proteção é feita com dois registradores: **registrador base** (define o limite inferior da área de memória do processo) e **registrador limite** (que, somado ao base, define o limite superior). Quando a CPU solicita um endereço, a **MMU** verifica se o endereço está dentro do intervalo permitido. Se o endereço estiver dentro do intervalo, o acesso é permitido; caso contrário, ocorre um erro fatal.

## Proteção de Memória e Relocação

O esquema de proteção de memória impede que um programa em modo usuário modifique o código ou os dados do sistema operacional ou de outros processos. Para isso, o processo recebe dois registradores: o registrador base (que contém o endereço inicial de sua área de memória) e o registrador limite (que define o tamanho dessa área). O Sistema Operacional carrega e controla esses registradores, e é necessário estar em modo Kernel para modificar esses valores. Isso garante que um processo de usuário só possa acessar sua própria área de memória, evitando que acesse a memória de outros processos.



O Sistema Operacional em modo Kernel pode acessar qualquer endereço de memória, o que permite operações como carregamento e descarregamento de programas, acesso e alteração de argumentos de chamadas de sistema, e a execução de operações de entrada e saída. Em sistemas multiprocessados, a troca de contexto é realizada ao transferir o estado de um processo dos registradores para a memória principal e carregar o contexto do próximo processo.

O processo pode ser alocado inicialmente em um endereço específico, mas, ao ser desalocado, pode ser realocado em outro endereço. Esse processo é chamado de **relocação** e é possível dependendo da política de alocação adotada. O processo de relocação será detalhado em módulos posteriores.

## Relocação de Memória

Quando um processo é alocado na memória principal, ele faz referência a endereços físicos específicos. Se esses endereços já estão definidos, o processo tem **endereçamento absoluto**. No entanto, para permitir que o processo seja alocado em qualquer lugar da memória, é necessário realizar a **relocação**, ou seja, mapear endereços lógicos (referenciados pelo processo) para endereços físicos (loais reais na memória).

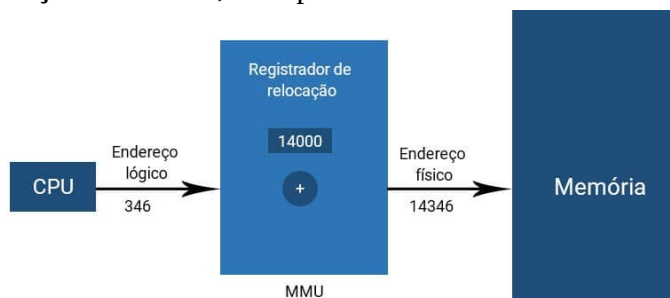
O **Espaço de Endereçamento** de um processo inclui:

- **Espaço Lógico de Endereçamento:** Endereços lógicos referenciados.
- **Espaço Físico de Endereçamento:** Endereços físicos correspondentes.

A relocação pode ser feita pelo **link-editor** durante o processo de vinculação, onde os endereços são resolvidos em relação a uma base inicial. Alguns sistemas fazem essa tarefa no **carregador** (loader) ou **ligador-carregador** (link-loader) no momento da execução do processo.

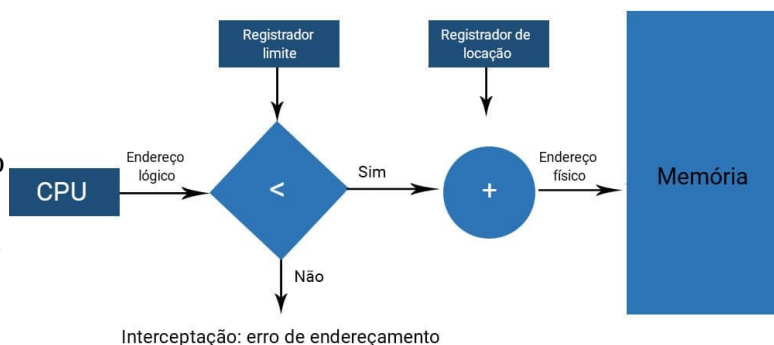
## Relocação Estática e Dinâmica

Na **relocação estática**, o mapeamento de endereços é feito antes do carregamento, e o processo sempre roda no mesmo lugar da memória. Já na **relocação dinâmica**, o mapeamento ocorre durante a execução, permitindo que o processo rode em qualquer lugar. Para a relocação dinâmica, o programa deve usar apenas endereços lógicos, pois a tradução para endereços físicos será feita pelo hardware. O registrador base, agora chamado **registrador de relocação**, contém o endereço inicial da área de memória do processo.



## Resumo: Relocação e Proteção de Memória

O registrador de relocação tem o valor 14000. Ao receber o endereço lógico 346, ele soma 14000, resultando no endereço 14346. A relocação permite que processos bloqueados sejam realocados em outros endereços quando executados novamente. Para a proteção de memória, verifica-se se o endereço lógico não ultrapassa o valor do registrador limite, evitando que o programa acesse além do limite superior da sua memória.



## Políticas de Alocação

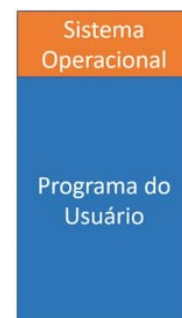
As políticas de alocação de memória são:

1. Manter os processos na memória principal durante toda a execução.
2. Mover os processos entre a memória principal e a secundária (disco) usando técnicas de swapping ou paginação.

## Gerenciamento de Memória sem Permuta

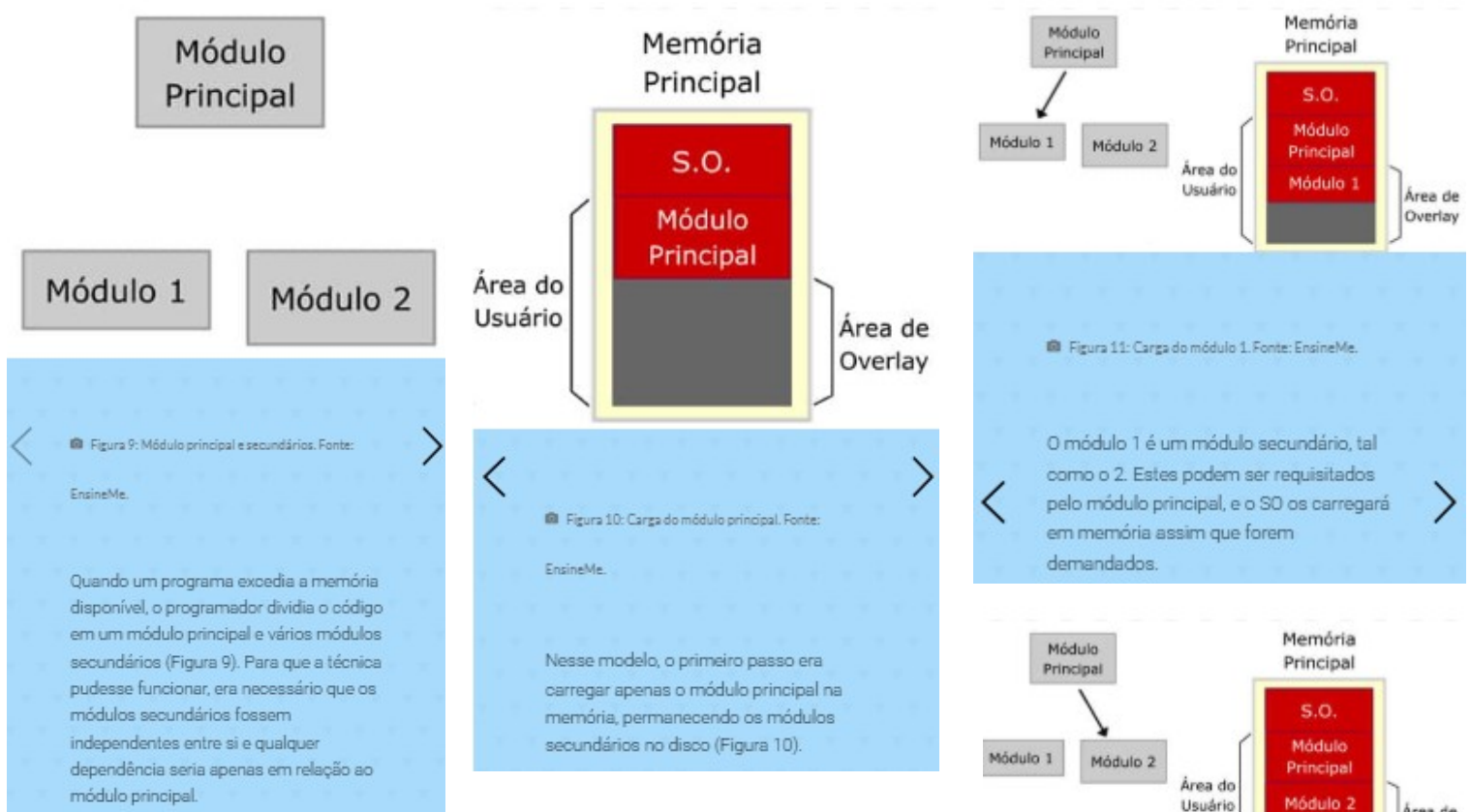
**Alocação Contígua:** Inicialmente, sistemas monoprogramados compartilhavam a memória entre o sistema operacional e o processo de usuário. Essa técnica limita o tamanho máximo do programa ao tamanho da memória disponível.

**Overlay:** Para superar essa limitação, foi criada a técnica de overlay, onde a mesma região de memória é ocupada por módulos diferentes do processo. O código é dividido em um módulo principal e vários módulos secundários, que são carregados na memória



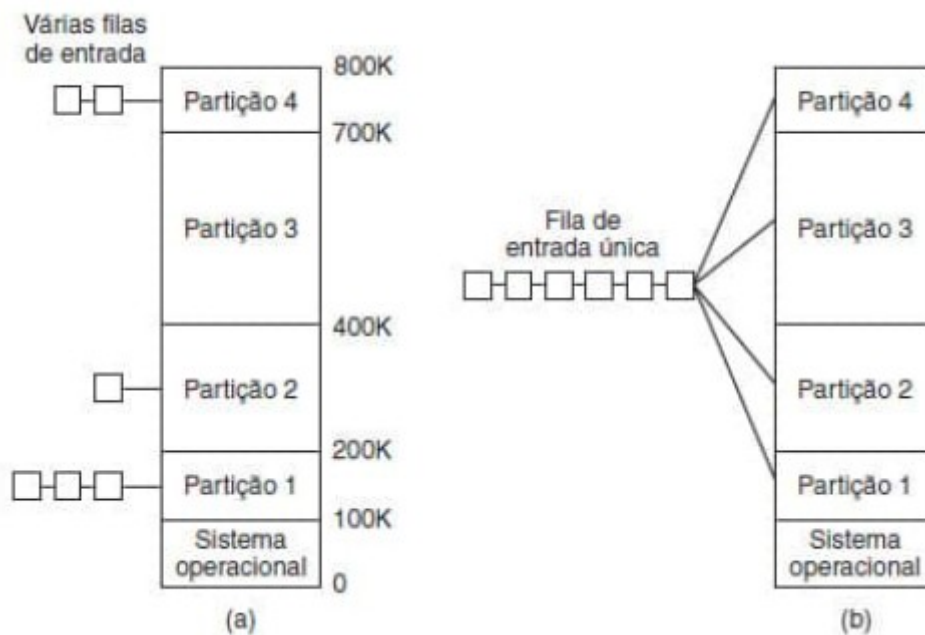
conforme necessário. O módulo principal permanece na memória, enquanto os módulos secundários ficam no disco e são carregados sob demanda.

O programador deve garantir o endereçamento correto e a execução adequada dos módulos.



## Alocação Particionada Fixa

Com a multitarefa, surgiu a necessidade de manter vários processos na memória simultaneamente. A solução foi dividir a memória em partições fixas de tamanhos diferentes, alocando um processo em cada partição. Essas partições eram predefinidas e o processo ficava na partição até seu término, liberando-a para outro processo na fila. Existem duas estratégias de alocação ilustradas na Figura 13.



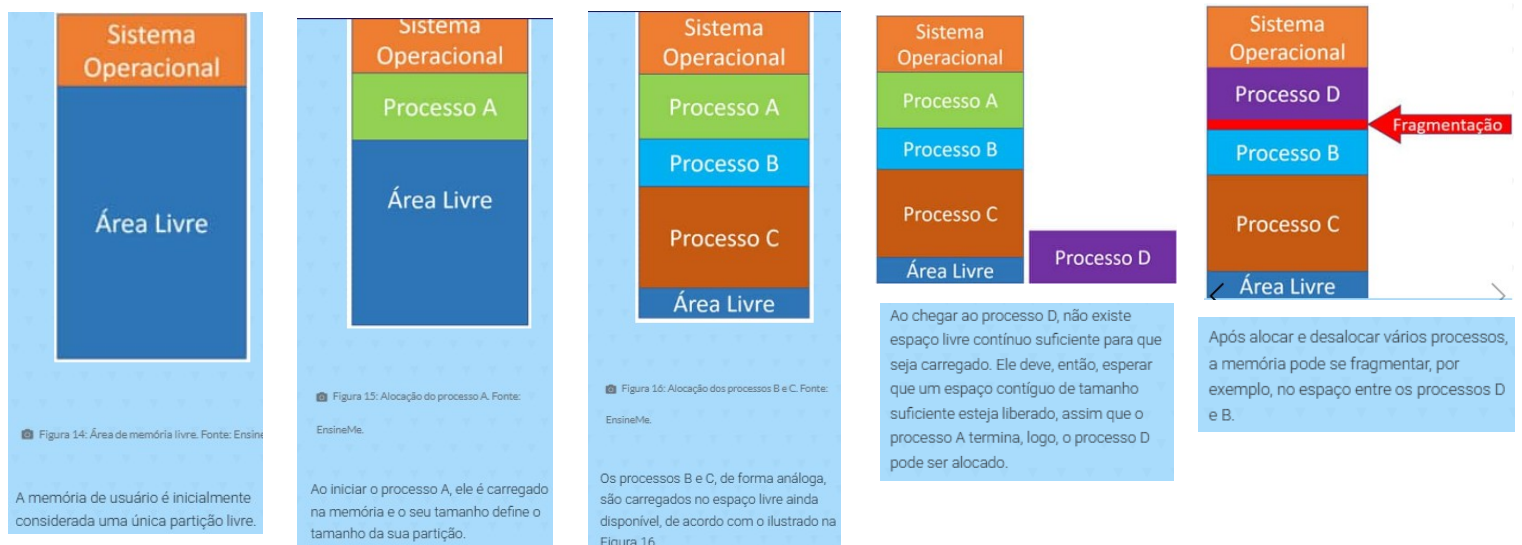
Existem duas estratégias de alocação de memória em partições fixas:

1. **Uma fila por partição:** os processos são alocados conforme o tamanho da partição disponível.
2. **Uma única fila de entrada:** os processos ficam na mesma fila e são alocados na menor partição que os acomode.

Esse método gera desperdício de memória, pois o espaço não utilizado em uma partição não pode ser aproveitado por outros processos. Para evitar isso, foi criado o esquema de alocação com partições variáveis.

## Alocação com partições variáveis

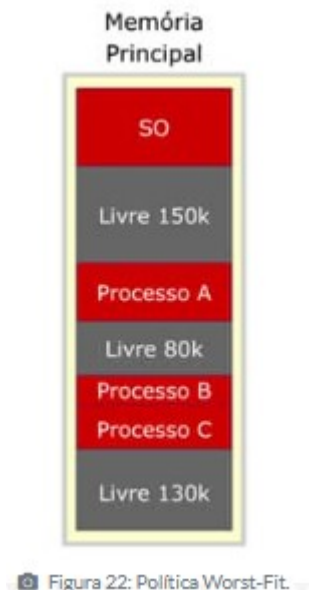
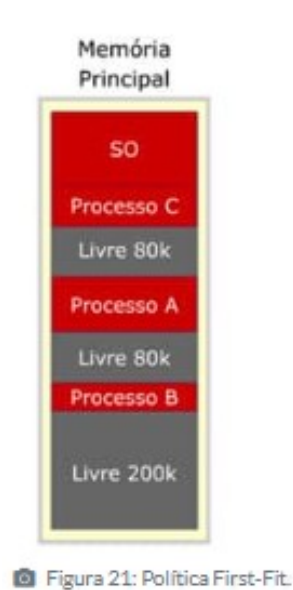
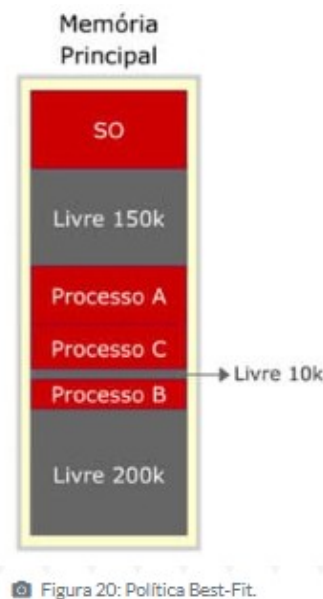
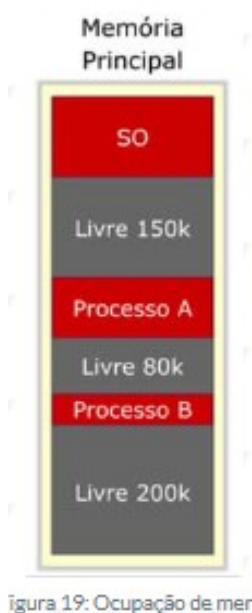
Nesta técnica, em vez de particionar a memória em blocos de tamanhos predeterminados...





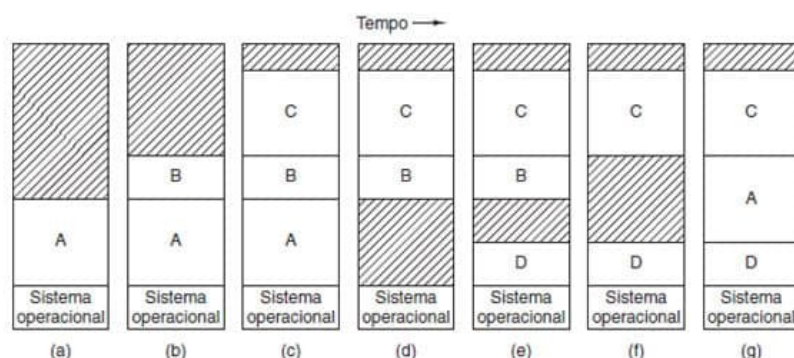
A escolha da partição para a próxima tarefa na fila pode ser feita por diversas políticas. Por exemplo, se um processo C de 70k precisa ser alocado, diferentes métodos podem ser utilizados para determinar a partição adequada.(19)

- **Política Best-Fit:** Aloca o processo na partição mais próxima do seu tamanho, gerando fragmentos menores, mas exige percorrer todas as partições. (20)
- **Política First-Fit:** Aloca o processo na primeira partição onde caiba, sendo mais rápida, mas pode agrupar processos pequenos. (21)
- **Política Worst-Fit:** Aloca o processo na maior partição disponível, gerando fragmentos grandes, mas precisa percorrer todas as partições. (22)



## Fragmentação externa e interna

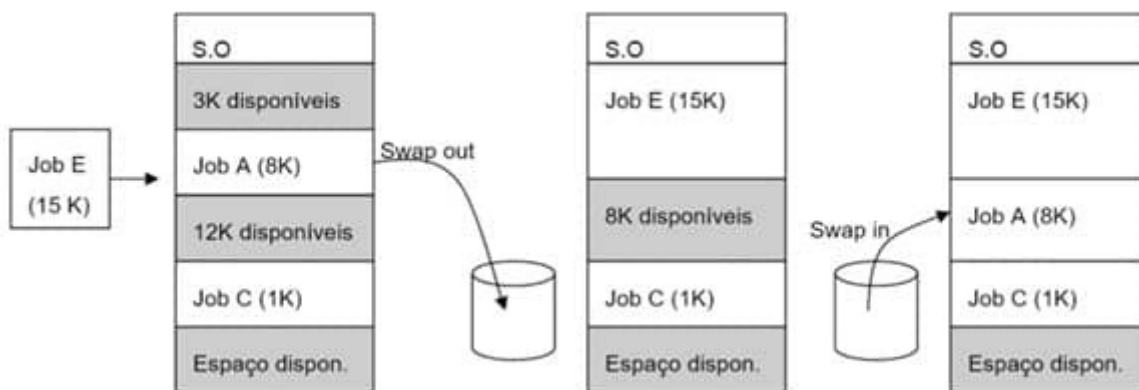
- **Fragmentação Externa:** Ocorre quando há pequenos espaços livres entre as partições, que, embora somados, não são contíguos, impedindo a alocação.
- **Fragmentação Interna:** Acontece quando há um espaço livre dentro de uma partição alocada, que não pode ser usado por outro processo.
- **Soluções:**
  - **Compactação:** Move blocos ocupados para uma extremidade, criando um grande bloco livre, mas é cara em termos de processamento.
  - **Endereçamento Não Contíguo:** Permite que o processo receba blocos de memória não contíguos, base da paginação e segmentação.



## Gerenciamento de Memória com Permuta

O **swapping** permite que um processo, quando está bloqueado ou aguardando, seja retirado da memória principal e transferido para a memória secundária, para ser recarregado quando sua vez de execução chegar. Isso aumenta a multiprogramação, pois possibilita que a memória total dos processos ultrapasse a memória física disponível, algo inviável com partições fixas.

No esquema de partições, um processo precisaria esperar até que um outro terminasse para liberar a memória. Com o swapping, se um processo (por exemplo, o processo A) não está em execução, ele pode ser "trocado" para o disco (swap out), liberando espaço para um processo aguardando (como o processo E) ser alocado na partição livre. Posteriormente, o processo que foi retirado pode ser recarregado (swap in) em outra partição disponível.



## Gerenciamento de Memória com Swap

Essa técnica utiliza relocação dinâmica, impedindo o uso de endereçamento absoluto pelos processos. Suas principais vantagens são:

- **Maior compartilhamento da memória:** Aumenta o throughput de tarefas.
- **Menor fragmentação de memória.**
- **Adequada para ambientes com processos pequenos e poucos usuários.**

A desvantagem é o tempo gasto no swap in e swap out. Por exemplo, se um processo de 200MB é removido (swap out) e outro de 150MB é carregado (swap in), com um HD que escreve a 50MB/s e lê a 100MB/s, o swap out leva 4 segundos e o swap in 1,5 segundos, totalizando 5,5 segundos – um tempo significativo em termos computacionais.

Devido a essa lentidão, a técnica de swap não é utilizada em sistemas modernos, que preferem métodos otimizados, como a paginação.

## Gerenciamento de Espaço Livre

Para implementar as políticas de alocação, o sistema operacional deve controlar os espaços livres e as áreas ocupadas na memória. Uma técnica comum é o uso de **mapas de bits**, onde a memória é

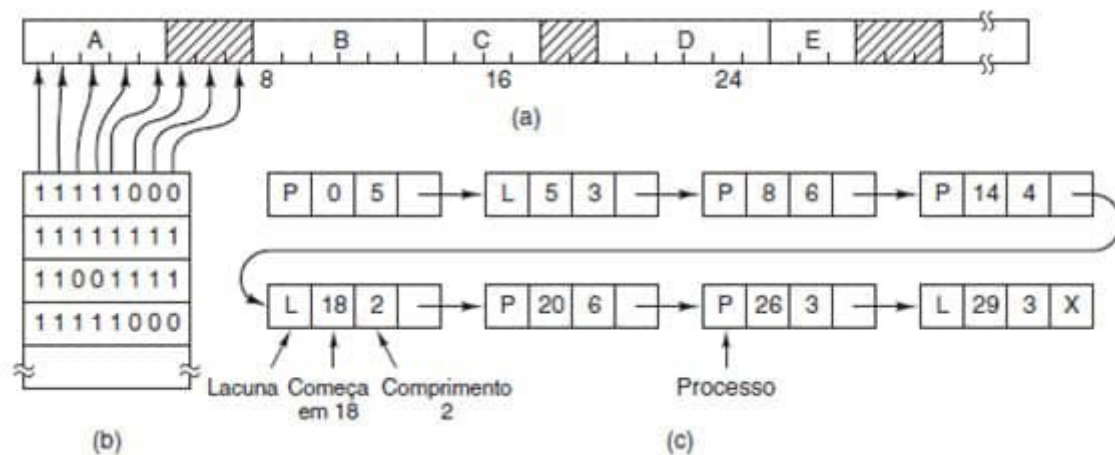
dividida em unidades de alocação e cada unidade é representada por um bit: 0 indica livre e 1 ocupado.

- **Tamanho da unidade:**

- Unidades menores geram mapas maiores.
- Unidades grandes podem causar fragmentação interna se o processo não preencher exatamente a unidade.

- **Desvantagem:**

- Para alocar um processo que necessita de  $n$  unidades, o gerenciador deve encontrar uma sequência de  $n$  bits 0 consecutivos, o que pode ser uma operação lenta.



## Gerência de Espaço Livre com Listas Encadeadas

A Figura 25 ilustra três representações da memória:

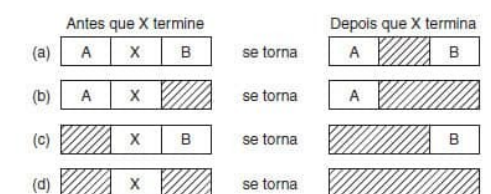
- (a) mostra uma parte da memória com cinco processos e três lacunas (unidades de alocação indicadas por traços).
- (b) apresenta o mapa de bits correspondente.
- (c) exibe as mesmas informações em uma lista encadeada.

Na **gerência de memória com listas encadeadas**, cada nó representa um segmento de memória, marcado como:

- **P:** segmento ocupado (processo)
- **L:** segmento livre

Cada nó armazena o endereço inicial e o tamanho do segmento, e a lista é ordenada por endereço. Quando um processo termina, a lista deve ser atualizada, podendo ocorrer:

- (a) Substituição de um nó ocupado por um livre.
- (b) e (c) Agregação de dois nós adjacentes em um único nó.
- (d) Agregação de três nós, removendo dois itens da lista.



Essa abordagem permite gerenciar dinamicamente os espaços livres na memória.

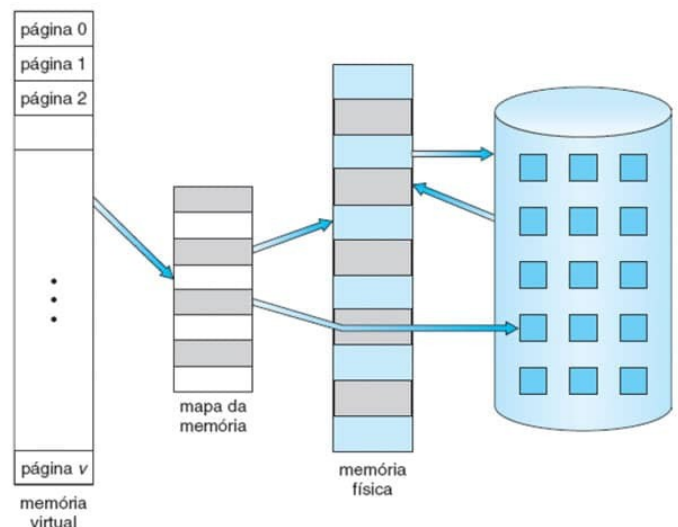
## Memória Virtual

O espaço de endereçamento lógico de um programa é mapeado para um espaço físico de memória, tarefa gerenciada pelo SO sem envolvimento do programa. O **espaço de endereçamento virtual** pode ser maior que a memória física disponível, permitindo que apenas partes do programa sejam carregadas conforme necessário.

Esse processo é transparente para usuários, compiladores e link-editores, sendo responsabilidade do SO gerenciar o carregamento e descarregamento dos blocos necessários. Como resultado, programas não precisam estar em regiões contíguas de memória.

A memória (virtual e real) é dividida em blocos, e o SO usa tabelas de mapeamento para associar os blocos da memória virtual de um processo aos da memória real da máquina, otimizando o desempenho.

A paginação permite que o espaço de endereçamento virtual seja maior que a memória física. O processo é dividido em páginas, que são transferidas entre disco e memória conforme necessário, implementando a ideia de swap.

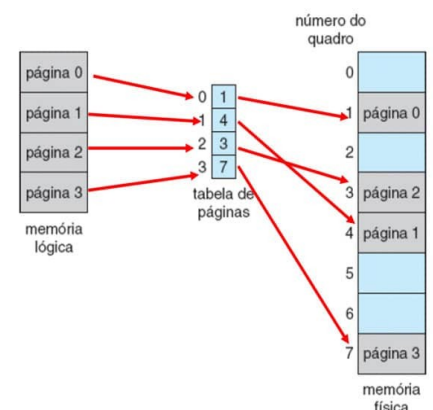


## Paginação

A paginação divide o espaço de endereços em páginas de tamanho fixo e a memória principal em molduras do mesmo tamanho. Os processos utilizam algumas páginas ativas na memória principal, enquanto as inativas permanecem na secundária. O sistema gerencia esse processo por meio de duas funções principais:

1. **Mapeamento** – Determina em qual bloco de memória a página referenciada está armazenada.
2. **Transferência** – Move páginas entre memória principal e secundária conforme necessário.

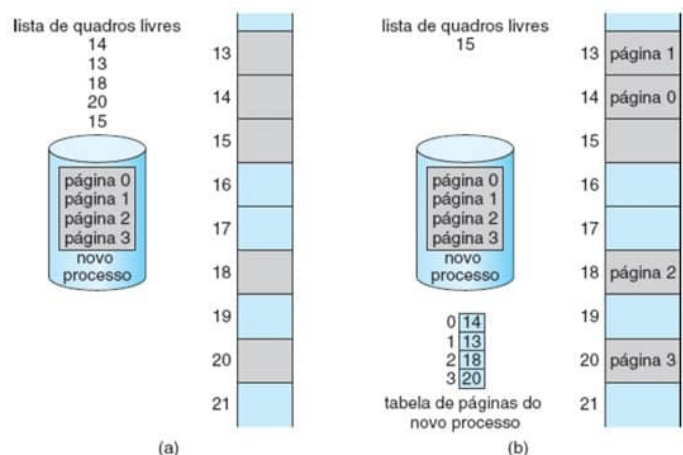
O hardware define o tamanho das páginas, que sempre é uma potência de 2 (normalmente entre 512 bytes e 2 KB), o que facilita o mapeamento entre endereços virtuais e físicos.



## Controle de espaço livre

O SO controla o espaço livre na memória mantendo uma lista de quadros disponíveis. Como todos os quadros têm o mesmo tamanho e a alocação não precisa ser contígua, essa lista não precisa ser ordenada, facilitando a manutenção.

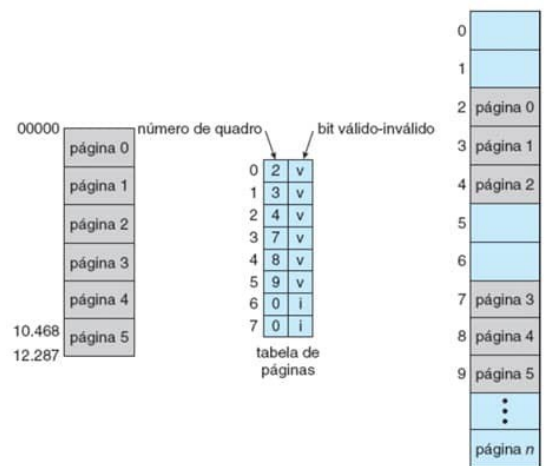
Quando um novo processo chega, seu tamanho é analisado em páginas, e cada página precisa de um quadro livre. Por exemplo, um processo com 4 páginas requer 4 quadros disponíveis. O SO aloca as páginas do processo nos primeiros quadros livres da lista, preenchendo sua tabela de páginas.



## Proteção de Memória

Em um sistema paginado, a proteção de memória é implementada com bits de proteção na tabela de páginas. Um bit define se a página é somente leitura ou permite gravação. Além disso, um bit válido-inválido indica se a página pertence ao espaço de endereçamento lógico do programa (acesso permitido) ou não (acesso proibido).

No exemplo da Figura 30, um sistema com endereçamento de 14 bits (0 a 16383) executa um processo que usa apenas os endereços de 0 a 10468. Com páginas de 2KB, o processo tem 6 páginas, embora a tabela suporte mais entradas. As páginas 6 e 7 têm número de quadro 0, o que resulta em uma exceção ao SO se referenciadas.

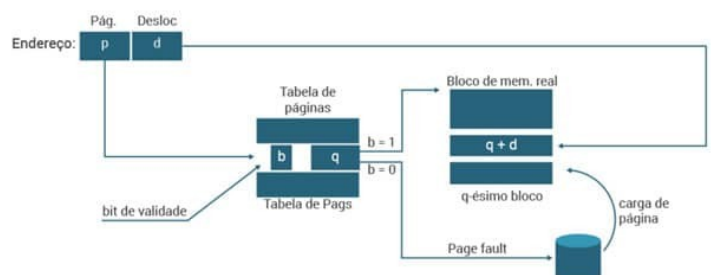


## Paginação Sob Demanda

A paginação sob demanda carrega na memória apenas as páginas referenciadas durante a execução. Inicialmente, apenas a página do processo é carregada. À medida que o programa acessa outras páginas, elas são carregadas conforme necessário. Páginas que nunca são referenciadas não são carregadas.

O número de blocos alocados a um processo é menor que o número de páginas que ele utiliza. Por isso, pode ocorrer que um endereço de programa referencie uma página ausente. Quando isso acontece, a entrada da tabela de páginas correspondente está "vazia", gerando uma interrupção do tipo *page fault*.

Essa interrupção aciona o processo de transferência da página ausente da memória secundária para a memória principal, e a tabela de páginas é atualizada. O processo fica bloqueado até a transferência ser concluída. A posição da página na memória secundária é registrada em uma tabela separada ou na própria tabela de páginas. Se não houver espaço na memória principal, um bloco de memória é desocupado conforme o algoritmo de troca.



## Políticas de Paginação

O conceito de *working set* foi criado para reduzir *page faults*. Quando um programa começa a executar, a probabilidade de acessar páginas ausentes na memória é alta, mas diminui à medida que mais páginas são carregadas, devido ao princípio da localidade dos programas. O SO define um conjunto de páginas a serem mantidas carregadas, equilibrando capacidade e velocidade.

Quanto mais páginas no *working set*, menor a chance de *page fault* e melhor o desempenho. Porém, mais páginas carregadas significam menos processos simultâneos na memória.

Ao ocorrer um *page fault*, o SO precisa decidir qual página descartar. Antes de remover qualquer página, o SO verifica se houve alteração nela. Se houve, a página é salva em disco, utilizando o Bit de Modificação na tabela de páginas para indicar se foi modificada.

## Políticas de Liberação de Páginas

As principais políticas de liberação de páginas são:

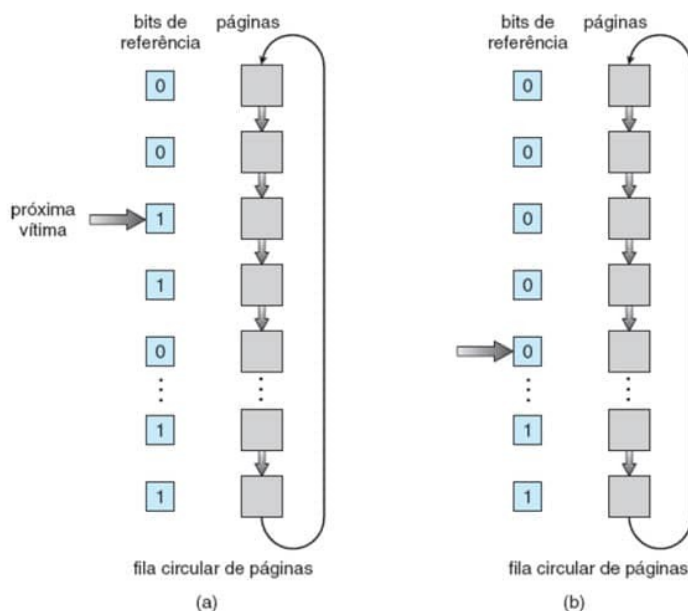
- **Página aleatória:** Escolhe qualquer página, o que pode resultar na remoção de páginas frequentemente acessadas.
- **First-in-first-out (FIFO):** Remove a página mais antiga, mas pode retirar páginas que são acessadas periodicamente.
- **Least-recently-used (LRU):** Remove a página menos recentemente acessada, mas causa overhead ao atualizar o momento de acesso a cada acesso.
- **Not-recently-used (NRU):** Substitui a página não utilizada nos últimos  $k$  acessos, também gerando overhead devido ao contador de acessos.
- **Least-frequently-used (LFU):** Retira a página com menos acessos, mas pode remover páginas recém-carregadas que têm poucos acessos iniciais.

Sistemas como o Linux costumam implementar variações da LRU, como o algoritmo de segunda chance.

## Algoritmo de Segunda Chance

O algoritmo de segunda chance, ou algoritmo do relógio, é uma aproximação do LRU. Cada entrada da tabela de páginas tem um bit de referência, inicialmente desligado. Quando uma página é referenciada, o bit é ativado. O sistema pode determinar se as páginas foram usadas examinando esses bits.

A política de substituição é FIFO, mas com uma diferença: antes de substituir uma página, verifica-se seu bit de referência. Se for 0, a página é substituída; se for 1, o bit é zerado e a





página recebe uma segunda chance. Isso garante que páginas frequentemente acessadas não sejam removidas.

O algoritmo é implementado com uma fila circular, com um ponteiro que indica a próxima página a ser substituída. Quando um quadro é necessário, o ponteiro avança até encontrar uma página com bit de referência 0. Se todos os bits estiverem ligados, o ponteiro circula a fila inteira, zerando os bits de referência e retornando à substituição FIFO.

## Aperfeiçoamento do Algoritmo de Segunda Chance

O algoritmo pode ser aprimorado ao considerar o bit de modificação da página, classificando-a em quatro categorias:

1. **(0, 0)** - Não usada nem modificada: Melhor opção para substituição.
2. **(0, 1)** - Não usada, mas modificada: Precisa ser gravada em disco antes de ser substituída.
3. **(1, 0)** - Usada recentemente, não modificada: Provavelmente será acessada novamente.
4. **(1, 1)** - Usada recentemente e modificada: Necessita ser gravada em disco antes de ser substituída.

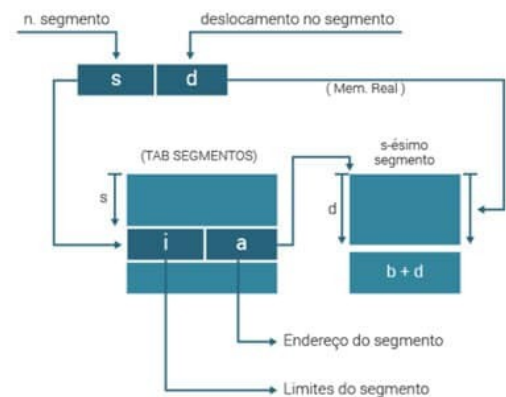
Nesse aprimoramento, usa-se uma fila circular, mas a substituição prefere páginas não modificadas para evitar gravações desnecessárias em disco. A substituição ocorre pela categoria mais baixa.

## Segmentação

A segmentação surgiu como alternativa à paginação para reduzir o número de *page faults*. Em vez de dividir o programa em blocos de tamanho fixo (páginas), a segmentação divide o programa em segmentos de tamanho variável com significado lógico. Isso evita a divisão física do programa, que pode causar muitos *page faults*.

A segmentação agrupa partes do programa que se referenciam mutuamente, evitando falhas de segmentação ao carregar essas partes juntas na memória.

O espaço de endereços torna-se bidimensional, representado pelo par (nome do segmento, endereço dentro do segmento). Para facilitar a transformação de endereços, o SO substitui o nome do segmento por um número ao referenciar o segmento pela primeira vez.



## Segmentação e Transformação de Endereço

A transformação de endereço é feita por meio de uma tabela de segmentos, com uma entrada para cada processo. Cada entrada contém o tamanho (l) e a posição inicial (a) do segmento na memória. As entradas são chamadas de Descritores de Segmento. O algoritmo de mapeamento é:

1. Extrair o endereço de programa (s,d).
2. Usar "s" para indexar a tabela de segmentos.
3. Obter o endereço inicial (a) do segmento.
4. Se  $d < 0$  ou  $d > l$ , ocorre uma violação de memória.

5. O endereço requerido é  $a + d$ .

Embora a busca de espaço para um segmento envolva problemas de gerência por partições variáveis, a diferença é que o espaço contíguo é necessário apenas para o segmento, não para o processo inteiro.

A principal diferença entre segmentação e paginação é que a segmentação divide o espaço de endereçamento logicamente, enquanto a paginação faz uma divisão física. Páginas têm tamanho fixo, enquanto segmentos variam conforme a memória disponível.

## Gerenciamento de Memória no Linux

O gerenciamento de memória no Linux envolve dois aspectos:

1. **Alocação e liberação de memória física** - Páginas, grupos de páginas e blocos de RAM.
2. **Manipulação da memória virtual** - Mapeamento da memória física para o espaço de endereçamento dos processos.

### Gerenciamento da memória física no Linux

O Linux divide a memória física em quatro zonas, que variam conforme a arquitetura. Para a arquitetura Intel x86 de 32 bits, as zonas são:

- **ZONE\_DMA**: Para dispositivos ISA que acessam os primeiros 16MB da memória.
- **ZONE\_DMA32**: Para dispositivos que acessam até 4GB da memória em operações DMA.
- **ZONE\_HIGHMEM**: Memória não mapeada para o kernel, nos primeiros 896MB do espaço de endereçamento.
- **ZONE\_NORMAL**: Todo o restante da memória.

Em sistemas modernos de 64 bits, como o Intel x86 de 64 bits, existe uma pequena ZONE\_DMA de 16MB, e o restante fica na ZONE\_NORMAL, sem ZONE\_HIGHMEM ou ZONE\_DMA32.

### Mapeamento das Zonas de Memória no Intel x86 (32 bits)

A Figura 34 mostra o mapeamento entre zonas e endereços físicos na arquitetura Intel x86 de 32 bits:

- **ZONE\_DMA**: < 16 MB
- **ZONE\_NORMAL**: 16 MB a 896 MB
- **ZONE\_HIGHMEM**: > 896 MB

O kernel mantém uma lista de páginas livres em cada zona e aloca páginas de acordo com a solicitação. Cada zona tem seu próprio alocador responsável pela alocação e desalocação de páginas físicas.

## Memória Virtual no Linux

O mecanismo de memória virtual gerencia o espaço de endereçamento de cada processo, mantendo duas visões:



- **Visão lógica:** Descreve as instruções relacionadas ao layout do espaço de endereçamento, com regiões não sobrepostas e contínuas de páginas. Essas regiões são organizadas em uma árvore binária balanceada para busca rápida.
- **Visão física:** Refere-se às entradas nas tabelas de páginas de hardware, que indicam a localização atual de cada página de memória virtual, seja em disco ou na memória física.

## Regiões de Memória Virtual no Linux

O Linux usa diferentes tipos de regiões de memória virtual, cada uma associada a um arquivo e funcionando como uma entrada para uma seção do arquivo de paginação. Quando o processo acessa uma página, a tabela de páginas é preenchida com o endereço da página do cache de páginas do kernel.

As regiões podem ser mapeadas como compartilhadas ou privadas:

- **Privada:** Gravações exigem uma cópia para manter as alterações locais.
- **Compartilhada:** Gravações atualizam o objeto mapeado, refletindo a mudança em outros processos que mapeiam a mesma região.

## Tempo de Vida de um Espaço de Endereçamento Virtual

Dois eventos geram a criação de um novo espaço de endereçamento virtual:

1. **Chamada de sistema `exec()`:** O processo recebe um espaço vazio e o mapeamento é feito com regiões de memória virtual.
2. **Chamada de sistema `fork()`:** O espaço de endereçamento do processo pai é copiado para o filho, compartilhando as mesmas páginas físicas de memória.

## Permuta e Paginação

No Linux, a substituição de páginas segue um processo em duas etapas:

1. Decisão sobre qual página substituir e se precisa ser gravada em disco.
2. Carregamento da nova página no quadro liberado.

O Linux utiliza uma política semelhante ao algoritmo do relógio, com múltiplos ciclos. Cada página tem uma "idade" que é ajustada a cada ciclo, refletindo o tempo de alocação e atividade recente. Páginas frequentemente acessadas têm uma idade maior, e as menos acessadas diminuem para zero. Esse mecanismo permite a seleção das páginas para remoção com base na política de "menos frequentemente utilizada" (LFU).

A memória virtual do kernel é reservada no espaço de endereçamento de cada processo e as entradas da tabela de páginas para essas regiões são protegidas para evitar modificações durante a execução do processo em modo usuário.

## Execução e Carga de Programas de Usuário

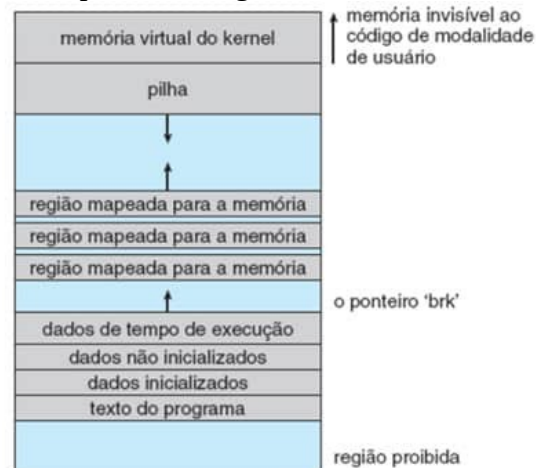
A execução de programas de usuário no Linux é iniciada pela chamada de sistema `exec()`, que substitui o contexto de execução atual pelo do novo programa. O carregador realiza o mapeamento do programa para a memória virtual e pode, ou não, alocar memória física para o processo.

## Mapeamento de Programas para a Memória

No Linux, o carregador binário não carrega diretamente um arquivo na memória física. Em vez disso, ele mapeia as páginas do arquivo binário para regiões de memória virtual usando a técnica de Paginação sob Demanda, alocando a memória física apenas quando necessário. O carregador ELF é responsável por estabelecer o mapeamento inicial da memória, lendo o cabeçalho do arquivo ELF e mapeando suas seções para regiões de memória virtual separadas.

A memória virtual é dividida entre o kernel, que ocupa uma região privilegiada e inacessível a processos de usuário, e as aplicações, que usam funções de mapeamento para criar regiões de memória. O carregador inicializa várias regiões, como a pilha, que cresce para baixo e inclui argumentos e variáveis de ambiente, e as seções de texto e dados do programa, que são mapeadas para a memória de forma protegida contra gravação.

O heap, uma área de tamanho variável que armazena dados em tempo de execução, é alocado após as seções de dados, com seu limite controlado pelo `brk`, permitindo sua expansão ou contração. Uma vez que os mapeamentos são estabelecidos, o carregador inicializa o registrador de contagem de programas com o ponto de início registrado no cabeçalho ELF, e o processo é então incluído no escalonador do sistema.



## Vinculação Estática e Dinâmica

Após o carregamento do programa, ele precisa de bibliotecas para sua execução. A vinculação pode ser estática ou dinâmica.

- **Vinculação Estática:** O código da biblioteca é incorporado diretamente ao executável pelo ligador, criando uma cópia exata das funções necessárias no programa. Embora isso permita a execução imediata, consome mais memória e espaço em disco, pois cada programa contém cópias das mesmas funções.
- **Vinculação Dinâmica:** Em vez de incluir as bibliotecas no executável, elas são carregadas apenas uma vez na memória. O programa contém uma função pequena que mapeia a biblioteca de vinculação para a memória ao iniciar. A biblioteca de vinculação determina e mapeia as bibliotecas necessárias para o programa, resolvendo as referências dos símbolos. As bibliotecas são compiladas como código independente de posição, permitindo que sejam executadas em qualquer local da memória.

## Utilitários e Comandos para Gerenciar a Memória do Sistema Linux

Diversos comandos e utilitários podem ser usados para gerenciar a memória no Linux. Um vídeo demonstrando esses processos será apresentado ao final do módulo.

### Obtendo informações pela linha de comando

```
ventury@ventury-VirtualBox:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.1 LTS
Release:        20.04
Codename:       focal
```

### Resumo: Comandos para Gerenciar a Memória no Linux

O Linux oferece diversos comandos para obter informações sobre a memória:

- **free:** Exibe o uso da memória, incluindo RAM e swap.

```
ventury@ventury-VirtualBox:~$ free -m
              total        usada       livre      compart.  buff/cache  disponível
Mem.:          3936          1235         213           89         2486         2338
Swap:           448           8         440
```

- **top:** Mostra processos e uso de memória.

```
ventury@ventury-VirtualBox:~$ top
top - 00:27:05 up 56 min, 1 user, load average: 0,06, 0,13, 0,28
Tarefas: 176 total, 1 em exec., 175 dormindo, 0 parado, 0 zumbi
%CPU(s): 7,1 us, 0,3 sis, 0,0 ni, 92,6 oc, 0,0 ag, 0,0 ih, 0,0 is 0,0 tr
MB mem : 3936,4 total, 213,9 livre, 1236,0 usados, 2486,5 buff/cache
MB swap: 448,5 total, 440,2 livre, 8,3 usados, 2338,9 mem dispon.

  PID  USUARIO  PR  NI  VIRT  RES  SHR  S  %CPU  %MEM  TEMPO+  COMANDO
    671  ventury   20   0 299596 110652 53100 S   3,3   2,7   2:47.25 Xorg
    988  ventury   20   0 3767984 432104 128412 S   2,7  10,7   3:24.88 gnome-+
  20304  ventury   20   0 828292 64564 47380 S   0,7   1,6   0:34.52 gnome-+
  50758  ventury   20   0 963324 52972 40168 S   0,3   1,3   0:01.06 gnome-+
     1   root    20   0 104332 13972 8512 S   0,0   0,3   0:04.52 systemd
     2   root    20   0      0      0      0 S   0,0   0,0   0:00.00 kthrea+
     3   root     0 -20      0      0      0 I   0,0   0,0   0:00.00 rcu_gp
     4   root     0 -20      0      0      0 I   0,0   0,0   0:00.00 rcu_pa+
     6   root     0 -20      0      0      0 I   0,0   0,0   0:00.00 kworke+
     9   root     0 -20      0      0      0 I   0,0   0,0   0:00.00 mm_per+
```

- **vmstat:** Exibe a situação da memória virtual.

```
ventury@ventury-VirtualBox:~$ vmstat
procs -----memória-----swap-- ----e/s---- -sistema- -----cpu-----
 r b  swpd  livre  buff  cache  si  so  bi  bo  in  cs  us  sy  id  wa  st
 2  0   9740 277436 156260 2309520  0  2  422 1040 359 1496 27  5 67  1  0
```

- **getconf PAGESIZE:** Mostra o tamanho da página de memória (4KB).

```

getconf PAGE_SIZE
ventury@ventury-VirtualBox:~$ getconf PAGE_SIZE
4096
ventury@ventury-VirtualBox:~$

```

- **swapon:** Exibe informações sobre o arquivo de swap.

```

ventury@ventury-VirtualBox:~$ swapon
NAME          TYPE      SIZE USED  PRIO
/swapfile     file     448,5M   9M   -2
ventury@ventury-VirtualBox:~$

```

## Utilitários para Acesso à Informação no Linux

Diversos utilitários monitoram o uso de memória no Linux:

- **Htop:** Evolução do comando top, exibe informações interativas. Deve ser instalado no Ubuntu.

```

ventury@ventury-VirtualBox:~$ htop

```

Após a chamada, ele carrega sua interface, na qual, em sua parte superior, temos informações agregadas e, na parte inferior, a lista de processos com suas informações de CPU, memória etc.

<div> <div>CPU[     ] 18.5%</div> <div>Mem[     ] 1.44G/3.84G</div> <div>Swp[ ] 10.3M/448M</div> </div> <div> <div>Tasks: 118, 265 thr; 1 running</div> <div>Load average: 0.37 0.24 0.18</div> <div>Uptime: 01:51:23</div> </div>											
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
671	ventury	20	0	366M	176M	87260	S	8.9	4.5	5:01.12	/usr/lib/
988	ventury	20	0	3735M	478M	154M	S	6.2	12.2	5:07.04	/usr/bin/
52002	ventury	20	0	766M	129M	99M	S	2.7	3.3	0:13.99	ksysguard
20304	ventury	20	0	809M	64812	47384	S	1.4	1.6	1:07.22	gnome-sys
50861	ventury	20	0	947M	56864	41124	S	1.4	1.4	0:05.10	/usr/libe
51979	ventury	20	0	10596	4000	3336	R	0.0	0.1	0:02.35	htop
761	ventury	20	0	366M	176M	87260	S	0.0	4.5	0:06.98	/usr/lib/
52008	ventury	20	0	4368	2496	2272	S	0.0	0.1	0:00.35	/usr/bin/
1375	ventury	20	0	486M	36092	26288	S	0.0	0.9	0:00.19	update-no
926	ventury	20	0	159M	6384	5736	S	0.0	0.2	0:00.18	/usr/libe
891	ventury	20	0	380M	9228	7076	S	0.0	0.2	0:02.13	/usr/bin/
898	ventury	20	0	372M	60856	39696	S	0.0	1.5	0:01.27	/usr/libe
1004	ventury	20	0	3735M	478M	154M	S	0.0	12.2	0:00.53	/usr/bin/



Neste aplicativo, as cores que aparecem no resumo na parte superior possuem significado.



## Indicadores de Uso no Sistema

- **CPU:**
  - Verde: Threads com prioridade normal.
  - Azul: Threads com baixa prioridade.
  - Vermelho: Threads em favor do kernel.
- **Memória:**
  - Verde: Memória usada pelas aplicações.
  - Azul: Buffers em uso.
  - Amarelo/Laranja: Cache.
- **Swap:**
  - Vermelha: Memória swap utilizada.

Essas informações também podem ser acessadas pressionando F1.

```
htop 2.2.0 - (C) 2004-2019 Hisham Muhammad
Released under the GNU GPL. See 'man' page for more info.

CPU usage bar: [low-priority/normal/kernel/virtualiz used%]
Memory bar:    [used/buffers/cache used/total]
Swap bar:      [used used/total]
Type and layout of header meters are configurable in the setup screen.

Status: R: running; S: sleeping; T: traced/stopped; Z: zombie; D: disk sl
Arrows: scroll process list          Space: tag process
Digits: incremental PID search      c: tag process and its child
F3 /: incremental name search       U: untag all processes
F4 \: incremental name filtering    F9 k: kill process/tagged proce
F5 t: tree view                    F7 ]: higher priority (root onl
p: toggle program path             F8 [: lower priority (+ nice)
u: show processes of a single user a: set CPU affinity
H: hide/show user process threads e: show process environment
K: hide/show kernel threads       i: set IO priority
```

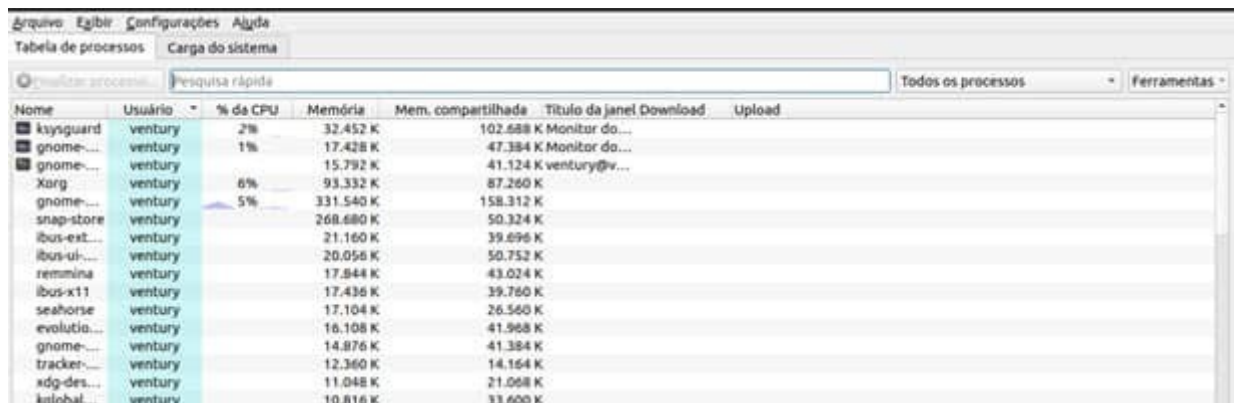
## Ksysguard

O utilitário ksysguard é outro que pode ser chamado na linha de comando e que não vem instalado por padrão.

```
ventury@ventury-VirtualBox:~$ ksysguard
Invalid Context= "Legacy" line for icon theme: "/usr/share/icons/Yaru/16x16/legacy
/"
```

Ele possui uma interface gráfica que mostra as informações do sistema.

Na aba Tabela de Processos, ele mostra todos os existentes no sistema e as informações de CPU e memória.



Nome	Usuário	% da CPU	Memória	Mem. compartilhada	Título da janela	Download	Upload
ksysguard	ventury	2%	32.452 K		102.688 K	Monitor do...	
gnome-...	ventury	1%	17.428 K		47.384 K	Monitor do...	
gnome-...	ventury		15.792 K		41.124 K	ventury@v...	
Xorg	ventury	6%	93.332 K		87.260 K		
gnome-...	ventury	5%	331.540 K		158.312 K		
snap-store	ventury		268.680 K		50.324 K		
ibus-ext...	ventury		21.160 K		39.696 K		
ibus-ui...	ventury		20.056 K		50.752 K		
remmina	ventury		17.844 K		43.024 K		
ibus-x11	ventury		17.436 K		39.760 K		
seahorse	ventury		17.104 K		26.560 K		
evolutio...	ventury		16.108 K		41.968 K		
gnome-...	ventury		14.876 K		41.384 K		
tracker	ventury		12.360 K		14.164 K		
xdg-des...	ventury		11.048 K		21.068 K		
kglobal...	ventury		10.816 K		33.600 K		

Na aba Carga do Sistema, ele mostra de forma gráfica a utilização pelos processos.



## Monitor do Sistema

Este aplicativo vem instalado por padrão e pode ser acessado na aba Ferramentas.



Após a abertura, ele apresenta uma interface gráfica com 3 abas na parte superior. São elas:

Processos								
Nome do processo	Usuário	% CPU	ID	Memória	Total de leitura	Total de escrita	Leitura no disco	Escrita no disco
at-spi2-registr...	ventury	0	926	648,0 KiB	100,0 KiB	N/D	N/D	N/D
at-spi-bus-launcher	ventury	0	915	944,0 KiB	48,0 KiB	N/D	N/D	N/D
bash	ventury	0	5065	248,0 KiB	4,0 KiB	N/D	N/D	N/D
bash	ventury	0	52013	1,4 MiB	432,0 KiB	48,0 KiB	N/D	N/D
dbus-daemon	ventury	0	673	2,8 MiB	828,0 KiB	N/D	N/D	N/D
dbus-daemon	ventury	0	920	452,0 KiB	20,0 KiB	N/D	N/D	N/D
dconf-service	ventury	0	979	744,0 KiB	92,0 KiB	412,0 KiB	N/D	N/D
evolution-addressbook-factory	ventury	0	1221	3,5 MiB	2,1 MiB	36,0 KiB	N/D	N/D
evolution-alarm-notify	ventury	0	1158	15,7 MiB	2,0 MiB	N/D	N/D	N/D
evolution-calendar-factory	ventury	0	1140	3,9 MiB	5,0 MiB	N/D	N/D	N/D
evolution-source-registry	ventury	0	1043	3,9 MiB	4,4 MiB	N/D	N/D	N/D



Vou enviar o conteúdo de uma página de uma apostila de estudo. Preciso de um resumo claro e direto. Eu não quero que você simplifique as coisas ao ponto de deixar o mínimo de informações possível, não é esse o objetivo aqui, o objetivo é pegar o texto original, e simplificá-lo, tirando quaisquer redundâncias, repetições, ou palavras desnecessárias. Se algo pode ser dito eficientemente com 5 palavras, não há necessidade de usar 10. Texto a ser resumido: