1 Introdução

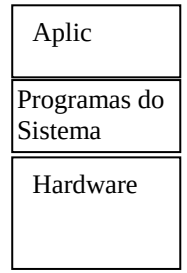
Sem o seu software, um computador é um amontoado de metal.
De todos os programas, o mais importante é o **Sistema Operacional**.
Computador moderno: um ou mais processadores, memória, relógio, terminais, discos, interfaces de rede, outros dispositivos de I/O.
Ou seja, um sistema complexo!
Escrever programas que trabalham com um ou mais desses elementos seria uma tarefa muito difícil. Se os programadores tivessem que se preocupar com os detalhes de funcionamento de cada disco, e com dúzias de coisas que podem dar errado ao se ler um bloco de disco, é provável que muitos programas não pudessem ser escritos.
Alguns anos atrás ficou claro que era necessário algum método para proteger o programador da complexidade do hardware. A forma encontrada foi colocar entre as aplicações e o hardware uma camada de software que serviria de **máquina virtual**: uma máquina mais fácil de entender e programar.
Esta camada de software é o sistema operacional.

Um computador é organizado em camadas.
A mais baixa é o **hardware**, composto de 2 ou 3 camadas:

- Dispositivos físicos, circuitos integrados, fios, fontes, etc.

Microprograma: em geral mantido em ROM, consiste de um interpretador que carrega instruções (ADD, MOVE, JUMP, etc.) e executa-as como um conjunto de passos.

<table>
<tr><td>Sistema Bancário</td><td>Reserva de voos</td><td>Jogos</td></tr>
<tr><td>Compiladores</td><td>Editores</td><td>Shell</td></tr>
<tr><td colspan="3">Sistema Operacional</td></tr>
<tr><td colspan="3">Linguagem de máquina</td></tr>
<tr><td colspan="3">Microprograma</td></tr>
<tr><td colspan="3">Dispositivos físicos</td></tr>
</table>



O conjunto de instruções define a **Linguagem de Máquina**.

A principal função do SO é esconder a complexidade do hardware e dar ao programador um conjunto de instruções mais apropriado. P. ex. é muito melhor escrever READ BLOCK FROM FILE do que se preocupar com mover as cabeças de leitura até a trilha certa, esperar que o bloco correto passe, fazer a leitura, etc.

O SO é aquela parte do sistema que executa em modo **protegido** (ou **kernel**, ou **supervisor**).

Acima do SO vem alguns programas como shell, compiladores, etc. Esses programas em geral são fornecidos pelo fabricante, mas não pertencem ao SO de fato, já que rodam em modo **usuário**.

Na última camada temos as aplicações do usuário.

1.1 O que é um SO?

1.1.1 O SO como uma máquina extendida

Ex. I/O numa unidade de disquete usando o controlador NEC PD765:

- Ele possui 16 comandos;
- Cada um carrega entre 1 e 9 bytes em registradores do dispositivo;
- Tipos de comandos: ler ou gravar dados, mover o braço, formatar trilhas, inicializar, status, reset, recalibrar a controladora, etc.
- Comandos mais básicos: READ e WRITE;
- Cada um precisa de 13 parâmetros, agrupados em 9 bytes;
- Eles especificam endereço de bloco, número de setores por trilha, modo de gravação, gap entre setores, o que fazer com marcas de endereços removidos.
- Quando a operação está pronta, a controladora devolve 23 campos de status e erro agrupados em 7 bytes;
- O programadore precisa se preocupar constantemente com o motor, se ele está ligado ou não;
- Se estiver desligado, precisa ser ligado (há um grande atraso inicial);
- O motor não pode ficar ligado mais que um certo tempo – ele pode queimar se esse tempo for ultrapassado;
- Assim, é preciso contar o tempo e desligar o motor nesse limite, independente do que ele estiver fazendo no momento – leva a perda de dados da operação corrente.

O que o programador precisa é uma abstração mais simples de trabalhar:

- O disco possui um conjunto de diretórios, cada um com um conjunto de arquivos;
- Uma vez aberto um arquivo, o programa pode ler e/ou gravar dados nesse arquivo;
- Após terminar suas operações o arquivo é fechado e as alterações são salvas.

Como o SO alcança este objetivo é uma longa história, que veremos em detalhe durante este curso.
Para resumir, o SO fornece uma série de serviços que os programadores podem usar através de instruções especiais chamadas **system calls**.

1.1.2 O SO como gerente de recursos

Ex.: suponha que 3 programas querem imprimir simultaneamente. Se não houver algum controle, a impressão pode resultar em algumas linhas de cada um por vez. Isso seria o caos.
Na verdade, o SO desvia a impressão de todos os processos para um buffer. Quando um programa termina, o SO copia sua saída de um arquivo em disco para a impressora. Ao mesmo tempo, os outros processos continuam imprimindo.

Processos podem interferir uns com os outros. Assim, o SO precisa tomar cuidados ao permitir o acesso aos seus recursos.
Ususários não compartilham apenas hardware, mas também informação (arquivos, BDs, etc.).

Assim, o SO precisa controlar quem acessa qual recurso, atender requisições dos processos, contabilizar o uso, mediar conflitos, etc.

O compartilhamento de recursos pode ser feito de duas formas:

- **Tempo**: os processos usam um recurso em turnos. P. ex.: CPU. O SO é responsável por determinar quem usa o recurso em um determinado momento e por quanto tempo.
- **Espaço**: ex.: memória. A memória pode ser dividida entre vários processos, que podem estar residentes ao mesmo tempo. Isso é mais eficiente do que dar toda a memória para um só processo, que pode precisar apenas de uma fração. É claro que isso leva a problemas de distribuição justa, proteção, etc. Mas é função do SO resolvê-los.

1.2 História dos sistemas operacionais

1o. computador digital de fato: "Máquina analítica" de Charles Babbage (1792-1871)
Não foi construído (não havia tecnologia para fazer essa máquina com engrenagens);
Não previa um SO;
Babbage concluiu que precisaria de software para seu projeto;
Convidou Ada Lovelace (filha de Lord Byron) para ser a 1ª programadora do mundo;
A linguagem Ada foi batizada em homenagem a ela.

1.2.1 Primeira geração (1945-55): válvulas e painéis de plugs

Na 2ª Guerra apareceram os primeiros resultados concretos de máquinas automáticas. Nos EUA, a 1ª máquina que é considerada o ancestral de todos os computadores foi o ENIAC (Electronic Numerical Integrator and Computer). Mas os ingleses tinham também sua máquina, o ACE (Authomatic Computer Engine), que ficou em segredo até a década de 70 e foi utilizado no esforço de quebrar o código do Enigma.
Por volta de 1945, o ENIAC possuía uma equipe que projetava, construía, programava, operava e dava manutenção. A programação era em linguagem de máquina usando um painel de plugs. Não havia linguagens de programação nem SO. A operação consistia em registrar num bloco a hora de início de uma tarefa, entrar na sala da máquina e introduzir o programa no painel, e esperar que nas próximas horas nenhuma das 20000 válvulas queimasse. As aplicações eram cálculos de tabelas como senos, cossenos e logaritmos, mas a máquina também era usada pela marinha para fazer cálculo de balística.
No começo dos anos 50 foi introduzido o cartão perfurado. Passou a ser possível guardar o programa em cartões e fazer a leitura deles quando necessário. O resto do procedimento era igual.
A equipe do ENIAC baseou seu trabalho nos estudos de Atanasoft, entre 1937 e 42. Esse trabalho introduziu alguns conceitos em uso ate hoje: aritmética binaria, memória regenerativa, lógica de circuitos, processamento paralelo e outros.
Entretanto, a base do sistema era um estudo de John von Neumann, que introduziu os conceitos de memória separada armazenando programas e dados, CPU e unidade de controle. Esse modelo deu origem a arquitetura de von Neumann, em uso até hoje. Von Neumann escreveu um artigo sobre sua proposta que nunca foi publicado por determinação do Depto. de Defesa, que considerou o trabalho secreto.

Alguns componentes da equipe do Eniac formaram uma empresa, a ECC (Electronic Controls Company), que fez uma proposta ao depto. do Censo para automatizar a tabulação dos dados coletados usando uma máquina derivada do ENIAC, o UNIVAC, entregue por volta de 1951. A ECC, renomeada para EMCC, foi vendida para a Remington Rand em 1950.

1.2.2 Segunda geração (1955-65): transistores e sistemas batch

O transistor mudou a situação radicalmente. A substituição das válvulas pelo transistor tornou os computadores confiáveis o suficiente para serem vendidos. O comprador tinha certeza de que a máquina funcionaria o suficiente para fazer um trabalho útil até o fim.
Permitiu a separação entre quem fabricava e quem operava.
Essas máquinas passaram a se chamar **mainframes**.
Somente grandes corporações, governos e universidades conseguiam pagar vários milhões de dólares para ter um.

Para rodar um **job**, o programador 1º. escrevia o programa em papel (em Fortran ou Assembler). Depois ele era perfurado em cartões. O **deck** era entregue a um operador e o resultado era retirado depois.
As salas eram divididas em Input, Processamento e Output.
Quando o computador terminava o que estava fazendo, o operador pegava um deck da sala de Input e passava numa leitora.
Se fosse necessário o compilador Fortran, o operador pegava o deck correspondente de uma prateleira e passava na leitora também.
Ao término de um job, o operador encaminhava seu resultado (impressão ou outro deck de cartão perfurado) à sala de Output.
Se perdia muito tempo de computador nesse trânsito entre salas.

Para reduzir o tempo perdido, foi criado o sistema **batch**.
A sala de Input passou a ter um computador menor (ex. IBM 1401) especializado em ler e perfurar cartões, e gravar fitas magnéticas.
O job recebido era agrupado com outros e transferido para uma fita magnética. Quando a fita estivesse cheia (ou após um tempo determinado – ex. 1 hora), ela era reenrolada e transferida para a sala de processamento.
A fita era montada em uma unidade de leitura e o operador carregava um programa especial (ancestral dos atuais SOs) para carregar job por job. A saída de uma execução era gravada em outra fita, que era encaminhada para a sala de Output, com outro IBM 1401 para imprimir.
Ao fim da fita de entrada, ela era retirada e substituída por uma nova fita batch.

Formato de um job:

- Cartão $JOB: continha o máximo de minutos que o job podia rodar, conta para contabilização e nome do programador;
- $FORTRAN: indicava ao operador que era preciso carregar o compilador Fortran;
- Programa a ser compilado;
- $LOAD: indicava que o programa compilado era para ser carregado na memória;
- $RUN: indicava ao SO para rodar o programa usando os dados a seguir;
- Dados do programa;
- $END: fim do job.

A principal tarefa da época era a solução de cálculos científicos e de engenharia (ex.: cálculo de equações diferenciais).

1.2.3 Terceira geração (1965-80): circuitos integrados e multiprogramação

No início dos anos 60, havia 2 linhas de computadores distintas (e incompatíveis):

- uma orientada a cálculos científicos e engenharia, composta de grandes máquinas (ex. IBM 7094);
- outra voltada a aplicações comerciais, como o 1401, usado para classificação de fitas e impressão em bancos e corretoras.

Desenvolver e manter essas duas linhas era caro para os fabricantes. Além disso, os clientes queriam comprar máquinas pequenas a princípio e evoluir para outras maiores.

A IBM tentou resolver essa situação introduzindo a linha System/360.
O 360 era uma linha de máquinas que ia de equivalentes ao 1401 até maiores que o 7094. A diferença era o preço e o desempenho (máximo de memória, velocidade de processamento, número de periféricos, etc.).
Todas elas tinham a mesma arquitetura e conjunto de instruções. Assim, teoricamente um programa escrito em uma delas poderia rodar em todas as outras.
O 360 foi desenhado para tratar aplicações científicas e comerciais.
Nos anos seguintes, a IBM lançou sucessores compatíveis com o 360: 370, 4300, 3080, e 3090.

O problema do 360 foi justamente sua proposta: funcionar da mesma forma para máquinas pequenas, com poucos periféricos e voltadas para leituras de fitas e impressão, e para grandes com muitos periféricos, voltadas para grandes jobs de alto processamento.

O resultado foi um SO bastante grande e complexo, cerca de 3 vezes maior que os SOs da época (cerca de milhões de linhas em assembler, mantido por milhares de programadores e apresentando enorme quantidade de bugs, o que obrigou a liberação de novas releases, que corrigiam alguns erros mas introduziam outros, de forma que o número de bugs deve ter-se mantido igual).

Apesar dos problemas, satisfez em grande parte as necessidades dos clientes. Também popularizou algumas técnicas ausentes na 2ª. geração.
A principal delas é a **multiprogramação**.
Antes, quando o job que estava rodando parava para pedir intervenção do operador ou fazia um I/O, a CPU ficava ociosa aguardando que a operação terminasse.
No 360, foi introduzido a divisão da memória em partições e sua ocupação por jobs diferentes que disputavam o processador. Quando um deles parava por algum motivo, outro assumia o processamento, economisando recursos da instalação.

Outra técnica interessante foi o **spool**.
Podia ser usado como input e output também. No input, permitia ler antecipado decks de cartão e jogá-los em disco. Quando fosse possível, o SO lia o job e processava sua execução.
Isso eliminou a necessidade de computadores de entrada.
No output, permitia jogar relatórios gerados pelos programas em disco para serem impressos depois. Isso eliminou a necessidade de computadores de saída.

Um problema sério que havia nos computadores da época era a forma de submissão de jobs através de equipes de preparação. O

processo levava várias horas. Um erro qualquer e um dia de trabalho podia ser desperdiçado.
Para resolver este outro problema, foi introduzido o **timesharing**. Consistia em permitir ao usuário acessar o sistema via um terminal. Em geral, nem todos os usuários estavam ativos ao mesmo tempo e isso permitia ao SO dividir a máquina entre os ativos, aceitando entradas de jobs de forma mais fácil e com mais rapidez.

Numa evolução do processo, o MIT, Bell Labs e General Electric se juntaram para desenvolver um sistema que teria capacidade para centenas de usuários timesharing. Surgiu o MULTICS (MULTiplexed Information and Computing Service). Para atender todos esses usuários, a máquina era apenas um pouco melhor que um PC Intel 386. A diferença era a capacidade de I/O.
Não chega a ser surpresa porque na época as pessoas sabiam como fazer programas pequenos e eficientes.
O MULTICS não conseguiu alcançar uma grande penetração no mercado por uma série de razões comerciais, entre elas porque era programado em PL/I.

Outro fato importante deste período foi o surgimento dos **minicomputadores**:

- DEC PDP-1 em 1961 – 4K palavras de 18 bits, $ 120.000, menos de 5% do custo de um 7094;
- Para certos tipos de tarefas não numéricas, apresentava desempenho equivalente a um 7094;
- Seguiu-se uma série de outros PDPs, incompatíveis entre si, até o PDP-11.

Unix:

- Desenvolvido por **Ken Thompson**;
- No Bell Labs, ele usou um PDP-7 esquecido e desenvolveu um novo sistema operacional;
- Para programá-lo, desenvolveu a linguagem B, precursora da linguagem C de Dennis Ritchie;
- Reescreveu o Unix em C após o seu lançamento;
- A grande vantagem do Unix era que todo o código era escrito em linguagem de alto nível;
- Apenas algumas partes hardware-específicas eram em assembler;
- Os fontes ficaram disponíveis e foram copiados por empresas e universidades, surgindo diversas cópias diferentes, incompatíveis entre si;
- Duas principais versões: System V da AT&T e BSD (Berkeley Software Distribution);
- O IEEE criou um padrão para o Unix chamado **Posix** para permitir o desenvolvimento de programas compatíveis entre as plataformas.

Tanenbaum desenvolveu um clone chamado **Minix** para fins acadêmicos.
Para finalidade geral, Linus Torvalds desenvolveu o **Linux**, baseado a princípio no Minix.

1.2.4 Quarta geração (1980): computadores pessoais

- LSI: Large Scale Integration
- 1974: Intel 8080
- Gary Kildall: CP/M (Control Program for Microcomputers) – Digital Research

- 1977: CP/M para Z80 e outros
- Bill Gates escreveu um interpretador BASIC, gravado em fita de papel
- Numa reunião de hobbistas, essa fita foi copiada (primeiro ato de pirataria conhecido)
- Bill Gates reclamou, alegando que se os programas fossem copiados dessa forma, os programadores não teriam retorno sobre seus produtos – não teriam interesse em desenvolver novos programas
- Ele criou o modelo de comercialização de software que vigora até hoje – o programa pertence à empresa desenvolvedora; os usuários adquirem licenças de uso
- 1980: a IBM projetou o IBM PC
- a IBM contactou Bill Gates para contratar seu interpretador BASIC
- a IBM consultou Bill Gates sobre um SO para o PC – ele indicou o CP/M
- Erros históricos: Kildall se recusou a conversar com a IBM diretamente – mandou um representante
- Seu advogado se recusou a assinar um documento garantindo sigilo sobre as discussões com a IBM
- A IBM voltou para Bill Gates e perguntou se ele podia arrumar um SO
- Na época, havia uma empresa (Seattle Computer Products) que tinha um SO apropriado – o DOS (Disk Operating System)
- Bill Gates comprou esse produto por $ 50.000,00
- Ele ofereceu à IBM um pacote com o DOS e o interpretador BASIC – a IBM aceitou
- Pediram a Bill Gates algumas modificações, que ele se comprometeu a fazer
- Para isso, contratou a pessoa que havia feito o DOS, Tim Paterson
- Bill Gates mudou o nome de sua companhia para Microsoft e chamou o SO de MS-DOS, que rapidamente dominou o mercado de IBM PCs
- A grande diferença de abordagem foi que a MS resolveu vender seu MS-DOS diretamente aos fabricantes de computadores para que fossem instalados de fábrica
- Kildall vendia o CP/M diretamente para o usuário final
- A Microsoft vendeu o MS-DOS em diferentes versões para os processadores que vieram depois: 80286, 80386 e 80486
- As últimas versões copiaram idéias do Unix
- A MS chegou a vender uma versão do Unix (Xenix)

- Em Stanford, foi criado o conceito de GUI (Graphical User Interface), que depois foi implementado pela Xerox
- Jobs e Wosniack (Apple) copiaram a idéia da Xerox em seu computador Lisa, que foi um fracasso
- Depois, apresentaram outro produto, o Macintosh, que foi um sucesso (user friendly)

- A MS copiou a idéia da Apple e lançou sua interface, o Windows
- O Windows 1.0 foi um fracasso, que desanimou Bill Gates a um ponto que ele ofereceu a MS para a IBM

- A IBM não quis porque não se interessava pelo software – achava que o mercado de computadores era definido pelo hardware
- A MS lançou o Windows 2.0, que foi um sucesso
- O Windows era um programa que rodava no topo do DOS
- Essa situação se manteve por 10 anos, até o lançamento do Windows 95
- Ele era um SO independente do MS-DOS, mas tinha ainda partes do DOS para boot e para rodar programas do MS-DOS
- Depois surgiu o Windows 98, também com uma boa porção de código 16 bits
- Em seguida, veio o Windows NT (New Technology), escrito do zero e totalmente 32 bits
- Seu objetivo era substituir as versões anteriores e o MS-DOS, mas não alcançou penetração até a versão 4.0
- O líder do projeto era David Cutler, um dos projetistas do SO do VAX, o VMS. Assim, o NT apresenta muitas idéias daquele produto
- Em 1999, a versão 5 do NT foi rebatizada de Windows 2000
- Uma nova versão do Windows 98 foi chamada de Windows Me (Millennium edition).

1.3 Tipos de sistemas operacionais

1.3.1 Mainframes

Distinguem-se dos computadores pessoais por sua capacidade de I/O.
Podem ter centenas de discos e centenas de terminais.
Atualmente ainda são usados como servidores Web, comércio eletrônico e transações B2B.
Tratam 3 tipos de serviços: batch, transações e timesharing.
Ex. OS/390.

1.3.2 Servidores

Servem múltiplos usuários simultâneos através de uma rede.
Permitem compartilhar software e hardware. Tipos de servidores: impressão, arquivos, Web, etc.
Ex. Unix, Windows 2000, Linux.

1.3.3 Multiprocessadores

Para máquinas com múltiplas CPUs.
De3pendendo da arquitetura, são chamados de computadores paralelos, multicomputadores ou multiprocessadores.
O SO é especial, mas em geral é um SO comum com alguns recursos especiais de comunicação e gerência.

1.3.4 Computadores pessoais

Apresentam boa interface para o usuário.
Principais aplicações: edição de texto, planilhas, Internet.
Ex. Windows 98 e 2000, Macintosh OS, Linux.

1.3.5 Tempo real

O principal parâmetro é o tempo.

Em geral, são usados em processos de controle industrial onde recebem dados e têm que atuar sobre certas máquinas.
Podem ser **hard real-time** (a ação **deve** ocorrer até determinado momento) ou soft real-time (a perda eventual de um deadline é aceitável).
Ex. VxWorks, QNX.

1.3.6 Embutidos

Usados em palmtops (ou PDA – Personal Digital Assistant) e sistemas embutidos.
Embutidos são sistemas que rodam em aparelhos de uso comum: TVs, microondas, celulares, etc.
Alguns têm características de tempo real, mas a principal característica é a restrição de recursos: memória, tamanho, energia.
Ex. PalmOS, Windows CE.

1.3.7 Smart Card

São cartões de crédito que contém uma CPU.
Apresentam severas restrições de memória e energia.
Podem tratar uma ou mais funções: pagamentos eletrônicos, etc.
Ainda não há padronização desses produtos.
Alguns modelos são orientados a Java. Apresentam uma JVM em ROM e carregam applets que são interpretados. Alguns tipos podem executar vários applets simultaneamente, levando a multiprocessamento e precisando de escalonamento. Também há necessidade de controle de recursos e proteção.

1.4 Revisão de hardware

O SO é intimamente relacionado com o hardware do computador. Ele estende o conjunto de instruções e gerencia os recursos da máquina.
Para funcionar, ele deve ter grande conhecimento do hardware do computador e de como ele parece para o programador.

1.4.1 Processador

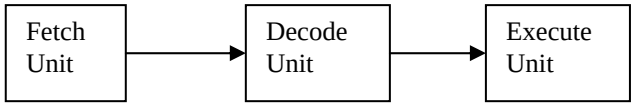
Busca instruções da memória e executa-as.
Cada CPU tem um conjunto específico de instruções (ex. um Pentium não pode executar programas SPARC).

As CPUs apresentam registradores especiais:
- **Contador de programa**: contém o endereço da próxima instrução;
- **Ponteiro de pilha**: aponta o topo da pilha do programa – esta pilha possui uma entrada para cada procedure chamada durante a execução;
- **PSW** (Program Status Word): contém os bits de condição, prioridade, modo (user ou kernel), etc.

Ao multiplexar a utilização da CPU, o SO deve salvar os registradores para o programa atual e restaurar os valores para o programa que for recomeçar.

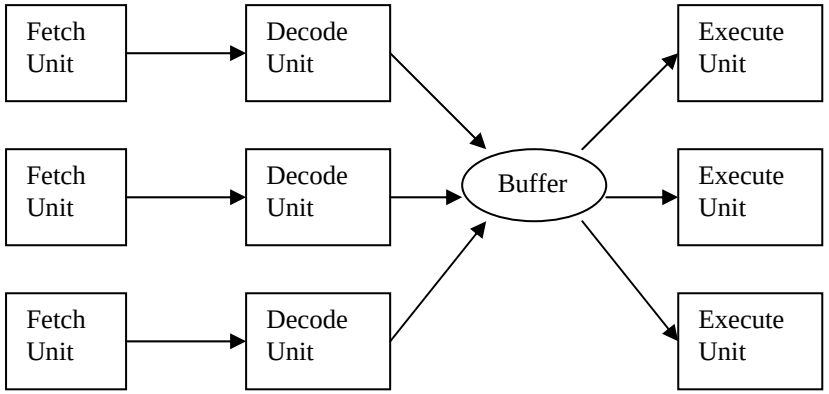
Para aumentar o desempenho, os projetistas lançam mão de alguns recursos bastante complexos.
Um deles é o **pipeline**.

P. ex.: em geral uma instrução apresenta várias fases: fetch, decodificação, execução.
Um pipeline é uma linha de produção, onde esses estágios são alinhados em unidades separadas.
Desse modo, o pipeline pode conter várias instruções diferentes, cada uma em um estágio diferente.
Ex.:



O ganho reside em que, ao final de cada estágio, uma instrução deixa o pipeline executada.
O uso de pipelines acelera a execução de instruções, mas apresenta alguns problemas. O principal deles são as instruções de jump condicional.
Quando um jump entra no pipeline, não se sabe se sua condição será verdadeira ou falsa. Assim, as instruções seguintes são carregadas sem se saber se o jump será executado ou não.
Se ele for executado, as instruções carregadas depois dele devem ser descartadas.
Os projetistas tentam achar formas de evitar os prejuizos que os jumps causam. Uma alternativa é associar com cada jump um valor – a chance do salto ser executado, dependendo de todas as vezes anteriores. Se a chance do salto ser executado for maior do que a chance dele não ser executado, o pipeline vai optar por fazer o fetch das instruções seguintes, como se o salto fosse feito.
Outra opção é o uso de pipelines duplos. Na presença de um salto, ambas as sequências seriam carregadas. Dependendo da condição do jump, um ou outro pipeline seria mantido. Isso só funciona para um jump por vez. Se houver outro jump antes do 1^{o}. ser executado, o pipeline deve optar por uma de suas continuações.

Uma forma + moderna de obter ganhos de desempenho é através do uso de arquiteturas **superescalares**.
Nessas arquiteturas, há múltiplas unidades de execução. Em geral, cada uma é especializada num tipo de instrução.



Assim, há unidades para aritmética inteira, de ponto flutuante, booleana, etc.
Há vários estágios de fetch-decode. Cada vez que uma instrução daquele tipo é encontrada, ela é introduzida na linha de fetch-decode apropriada.
Ao terminar o processamento dentro de uma linha, a instrução vai para um buffer especial.
Cada unidade de execução verifica o buffer quando estiver livre. Se houver uma instrução do tipo apropriado, ela é retirada do buffer e executada.

Esse sistema causa a execução das instruções fora de ordem. Em geral, o hardware é responsável por garantir que o resultado é o mesmo que uma execução sequencial. Mas ainda assim, uma parte da complexidade é de responsabilidade do compilador e do SO.

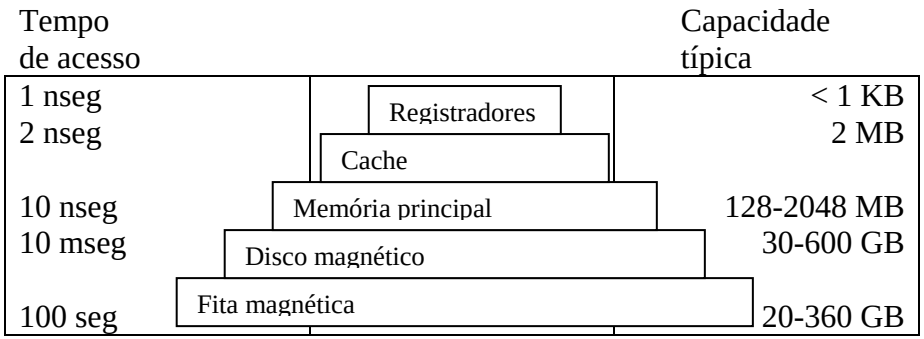
Executando no modo **kernel**, a CPU pode executar todas as instruções possíveis do conjunto de instruções do processador. Também pode acessar todos os recursos de hardware disponíveis.
No modo **user**, só um subconjunto de instruções pode ser executado. Somente poucos recursos de hardware podem ser acessados.

A passagem do modo user para o modo kernel só é possível através de uma interrupção.

Para fazer requisições ao SO, um programa usuário deve usar **system calls**. Cada uma usa uma interrupção por software (trap) para acionar o SO.

1.4.2 Memória

2º. maior componente de um sistema, a memória deveria ser extremamente rápida (assim não atrasaria a CPU), abundante e barata.
Na prática, nenhuma tecnologia satisfaz esses requisitos. Assim, é necessário jogar com vários tipos de memória para obter o resultado desejado.



A 1ª. camada consiste dos **registradores** internos. Eles são feitos do mesmo material da CPU e são tão rápidos quanto ela. Tipicamente, uma CPU de 32 bits possui 32 registradores de 32 bits; uma de 64 bits possui 64 de 64 bits.
Seu uso é definido por programa.

A memória **cache** é controlada por hardware.
Os blocos de memória principal mais utilizados pelo processador são mantidos no cache.
Quando um programa precisa de uma posição de memória, o hardware verifica se essa posição está em um dos blocos dentro do cache. Se estiver (um **cache hit**), ela é entregue a partir do cache com ganho de tempo (em torno de 2 ciclos). Se não, é feito acesso à memória, com custo bem maior.
Caches possuem tamanho limitado, devido ao seu alto custo.
Em geral, são organizados em hierarquias, onde os principais são pequenos mas muito rápidos, enquanto os secundários são maiores mas mais lentos.

Em seguida, vem a **memória principal**, também chamada de **RAM** (Random Access Memory).
Atualmente, a RAM apresenta tamanhos na casa dos 128 a 2048 MB, crescendo rápido. Todas os cache faults são dirigidos à RAM.

O próximo nível é o dos **discos**.
Os discos atualmente estão na casa das 500 vezes a capacidade da RAM; o preço dos bits em disco é bem menor que em RAM; em compensação, o tempo de acesso pode chegar à casa de 10^6 vezes superior ao acesso em RAM.
Um disco consiste de um conjunto de **pratos** que giram à velocidade de 5400, 7200 ou 10800 RPM.
Para cada prato, há uma **cabeça** de RW que pode acessar ambas as **superfícies** do prato. Os dados são gravados em círculos concêntricos, chamados de **trilhas**. O conjunto de trilhas em uma mesma posição do braço chama-se **cilindro**.
Cada trilha é dividida em **setores**; em geral, seu tamanho é de 512 bytes, mas pode ser de 2KB ou mais.
Nos discos modernos, as trilhas externas têm mais setores que as internas.
Mover o braço de um cilindro para o próximo leva cerca de 1 mseg. Para um cilindro aleatório, leva cerca de 5 a 10 msegs. Chegando à trilha correta, a espera pelo setor correto pode levar de 5 a 10 msegs, dependendo do RPM do disco.
Estando sobre o setor correto, a leitura/gravação ocorre a taxas entre 5 MB/seg (discos velhos) e 160 MB/seg (discos rápidos).

O último item da hierarquia são as **fitas magnéticas**.
Sua função é receber backups de discos ou armazenar arquivos muito grandes.
Para acessar uma fita, ela deve ser posta numa unidade de fita, por uma pessoa ou por um robo.
A seguir, é preciso pesquisar na fita pelo bloco desejado. Isso pode levar alguns minutos.
Sua grande vantagem é o preço reduzido por bit armazenado; além disso, pode ser removida e armazenada em lugares diferentes e distantes de onde está o computador.

Além desses tipos de memórias da hierarquia, um computador pode ter alguns tipos diferentes de memória principal:

- **ROM (Read Only Memory)**: rápida e barata, vem carregada de fábrica. Em geral, armazena programas de boot que iniciam o computador quando ele é ligado. Em cartões de I/O, armazena as rotinas de tratamento dos dispositivos.
- **EEPROM (Electrically Erasable ROM)**: não volátil, mas pode ser apagada e reescrita. Usadas como ROM, mas permitem a correção dos programas gravados.
- **Flash RAM**: não volátil; pode ser modificada pelo processador, apesar de que as operações são muito demoradas. Usada em dispositivos especiais, como calculadoras, para manter valores quando o dispositivo é desligado.
- **CMOS**: memória volátil de baixo consumo que pode ser mantida por bateria. Usada para manter configurações de hardware, data e hora, etc. Uma bateria pequena pode manter os dados por anos.

É desejável manter mais de um programa na memória principal.
Se um programa bloqueia para fazer um acesso a disco, outro pode usar a CPU resultando em melhor desempenho do sistema.

Há 2 problemas a resolver:

- Como **proteger** um programa do outro, e o Kernel de todos?
- Como **relocar** programas?

Qual a importância do 2º. problema?
Quando um programa é compilado e lincado, não se sabe em que posição da memória ele será carregado. Inclusive, essa posição pode variar de execução para execução, e até dentro da mesma execução. Assim, assume-se que qualquer programa vai executar na posição 0. Se o programa for carregado em uma posição diferente, uma possibilidade é varrer todo o código e alterar todas as instruções com endereços para acessar as posições a partir da nova **posição base**. Ex.: um programa de 50 KB carregado na posição 150 KB.

- Um endereço 10 KB deve ser traduzido para 160 KB.

Isto é possível, mas caro.
Uma solução melhor é acrescentar à CPU 2 registradores: **BASE** e **LIMIT**.
Quando um programa é executado, BASE é carregado com o endereço inicial onde o programa foi carregado (no ex. 150 KB). LIMIT é carregado com o endereço final do programa (no ex. 150 KB + 50 KB = 200 KB).
Quando o programa gera um endereço, ele é somado com o registrador BASE (no ex. 10 KB + 150 KB = 160 KB).
O resultado deve ser menor que o registrador LIMIT (160 KB < 200 KB = TRUE!).
Esse esquema resolve os 2 problemas: os programas podem ser relocados sem alteração do código e a proteção é garantida.

O custo total é: 2 registradores a mais na CPU; salvar esses 2 registradores para cada programa; fazer o teste do LIMIT para cada endereço.

2 aspectos que tem influência no desempenho de um sistema:
Primeiro:

- Caches escondem a velocidade relativamente baixa da memória principal;
- Quando um programa executa, o cache fica cheio com suas páginas + acessadas;
- Isso melhora o seu desempenho;
- Quando o SO troca o programa atual por outro, o cache mantém por algum tempo as páginas do 1º.;
- Assim, as páginas do novo programa devem ser acessadas na memória principal (ou mesmo de disco);
- Isso leva a uma queda de desempenho nos 1os. instantes após uma **troca de contexto**.

Segundo:

- As CPUs modernas têm um dispositivo chamado MMU;
- Esse dispositivo transforma endereços chamados **virtuais** em **reais**;
- Ela possui vários registradores e tabelas;
- Ela deve ser carregada para cada programa;
- Uma troca de contexto obriga salvar todo o conteúdo da MMU para o programa que sai e carregar todo o conteúdo para o programa que entra;
- Isso causa um grande overhead na troca de contexto.

1.4.3 Dispositivos de E/S

Um dispositivo é composto basicamente por duas partes:

- Controladora: em geral, uma placa de circuito eletrônico que controla o dispositivo. Ela aceita comandos do SO e comanda sua execução pelo dispositivo;
- Dispositivo: quem realmente realiza as atividades que o SO pede.

As vezes, o controle do dispositivo é bastante complexo. Assim, uma das tarefas da controladora é apresentar uma interface + simples para o SO.
P. ex.: num disco, o SO pede para ler o setor de número X.

- A controladora deve converter X para cilindro, cabeça e setor;
- Um complicador é o fato dos cilindros + externos terem mais setores que os internos;
- O disco pode remapear um bad sector para outro setor, em outro lugar do disco.

Em geral, as interfaces são simples. Dispositivos não fazem muita coisa. Além disso, precisam ser padronizados.
P. ex.: uma controladora IDE (Integrated Drive Eletronics) pode manipular qualquer disco IDE.

O software que conversa com a controladora, passando comandos e recebendo resultados, é chamado de **device driver**.
Cada fabricante de dispositivos deve fornecer os device drivers, um para cada SO suportado (Windows 98, Windows 2000, Windows XP, Unix, etc.).

Para executar, os drivers precisam fazer parte do SO. Assim, podem rodar em modo Kernel, onde tem privilégio para trocar msgs com as controladoras.
Há 3 formas de de se colocar um driver dentro do kernel:

- Relincar o kernel com o novo driver e dar boot no sistema – modo como se faz no Unix;
- Indicar que o SO necessita do driver e dar boot. Durante o boot, o SO localiza o driver e carrega-o – modo Windows;
- Indicar ao SO o novo driver, que é carregado sem a necessidade de boot – método pouco comum, mas tornado-se popular – usado em dispositivos USB, p. ex.

Há duas formas de arquitetura de I/O:

- **Mapeado em memória**: os registradores das controladoras são mapeados em endereços de memória. Mover dados para essas posições significa movê-los diretamente para a controladora.
    - Não há necessidade instruções especiais de I/O.
- **Portas de I/O**: cada registrador de uma controladora recebe um endereço de porta e instruções especiais IN e OUT permitem ler e gravar nos registradores.

Há três formas de se fazer I/O:

- **System calls**: o método + simples.
    - Um programa chama uma system call.
    - O Kernel traduz a chamada para uma chamada a um driver.
    - O driver dispara o I/O e entra em loop, testando se o pedido já foi feito.
    - Quando estiver, o driver coloca os dados onde foi pedido e retorna.

- O SO retorna o controle para quem chamou.
- Método chamado **busy waiting** (espera ocupada), porque gasta CPU enquanto aguarda.

- **Interrupções**:
  - O driver dispara o I/O e pede ao dispositivo que faça uma interrupção quando ele tiver terminado.
  - O SO bloqueia quem chamou e vai fazer outra coisa.
  - Quando a controladora detecta o fim do I/O, ela gera um sinal de interrupção para avisar.

- **DMA (Direct Memory Access)**:
  - Chip que controla o fluxo de transferência entre a controladora e a memória, sem intervenção da CPU.
  - A CPU informa ao chip quantos bytes transferir, dispositivo, endereços de memória e direção. O chip faz o resto.
  - Quando o I/O acaba, ele causa uma interrupção, que é tratada como acima.

Interrupções podem ocorrer em momentos inconvenientes.
Para evitar problemas, é possível desabilitar interrupções. Se uma é gerada nesse período, ela não é aceita até que as interrupções sejam habilitadas de novo. O dispositivo mantém o sinal enquanto a CPU não avisar que aceitou.
Caso vários dispositivos queiram gerar interrupções, é atribuído a cada um uma prioridade estática. A maior prioridade é atendida primeiro.

1.4.4 Buses

O IBM PC foi lançado com um único bus para todo o sistema.
Ã medida que os processadores e memórias se tornaram + rápidos, o bus único se tornou um gargalo para o sistema.
Hoje, vários buses diferentes foram adicionados e a velocidade de cada um aumentou muito:

- Cache, entre o cache e a CPU;
- Local, entre a CPU e a ponte PCI;
- Memória, entre a ponte PCI e a memória principal;
- PCI, entre a ponte PCI e vários dispositivos: SCSI, USB, adaptadores gráficos, ponte ISA, etc.
- ISA, entre a ponte ISA e diversos dispositivos: modem, placa de som, impressoras, etc.

Bus ISA (Industry Standard Architecture):

- Bus original do IBM PC/AT.
- Roda a 8.33 MHz e transfere 2 bytes de cada vez.
- Máximo de 16.67 MB/seg.

PCI (Peripheral Component Interconnect):

- Intel, successor do ISA;
- 66 MHz, transfere 8 bytes por vez;
- 528 MB/seg.

USB (Universal Serial Bus):

- 1.5 MB/seg;
- Um dispositivo raiz faz poll a cada mseg para todos os demais para verificar se há algum tráfego;

- Não há necessidade de drivers novos para cada dispositivo;
- Não é preciso boot após a instalação de um novo dispositivo.

SCSI (Small Computer System Interface):
- Barramento de alto desempenho para dispositivos que necessitam de banda larga;
- 160 MB/seg.

IEEE 1394 (FireWire):
- 50 MB/seg;
- Bus serial originário da Apple;
- Não há dispositivo central.

Plug and play:
- Sistema de interligação de dispositivos;
- Antes do plug and play, cada cartão tinha uma interrupção fixa e endereços fixos para os registradores de I/O;
- Ex.:
    - Teclado: interrupção 1, endereços 0x60 a 0x64;
    - Disquete: interrupção 6, endereços 0x3F0 a 0x3F7;
    - Impressora: interrupção 7, endereços 0x378 a 0x37A.

- Problema: o usuário comprou 2 cartões que usam a mesma interrupção;
- Para resolver isso, optou-se por colocar em cada cartão um sistema de switches para escolher as interrupções de cada um;
- O acerto de todos os dispositivos de um sistema não é tarefa trivial;
- Com plug and play, o sistema coleta as informações sobre todos os dispositivos, escolhe as melhores opções e informa aos cartões as interrupções com que eles vão trabalhar.