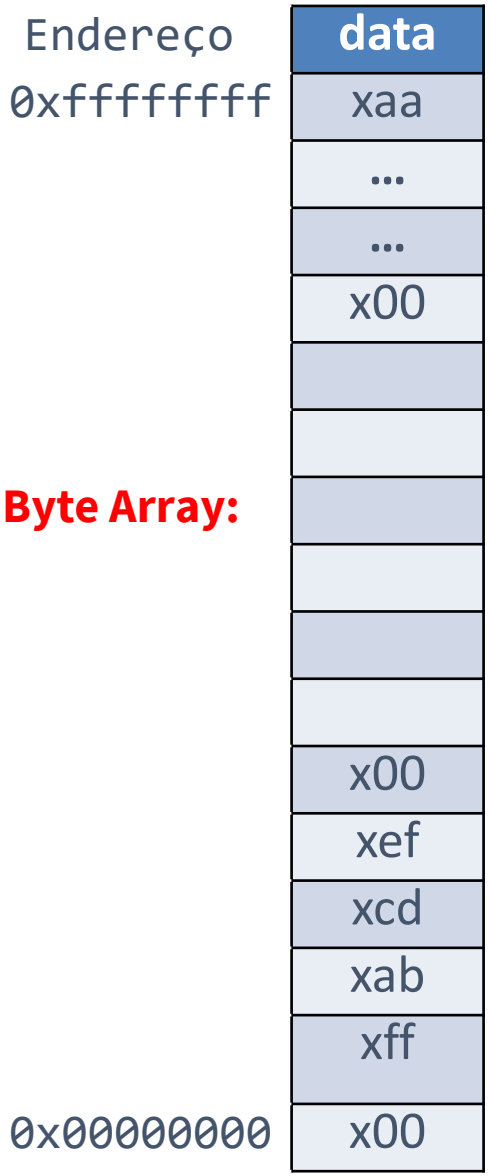
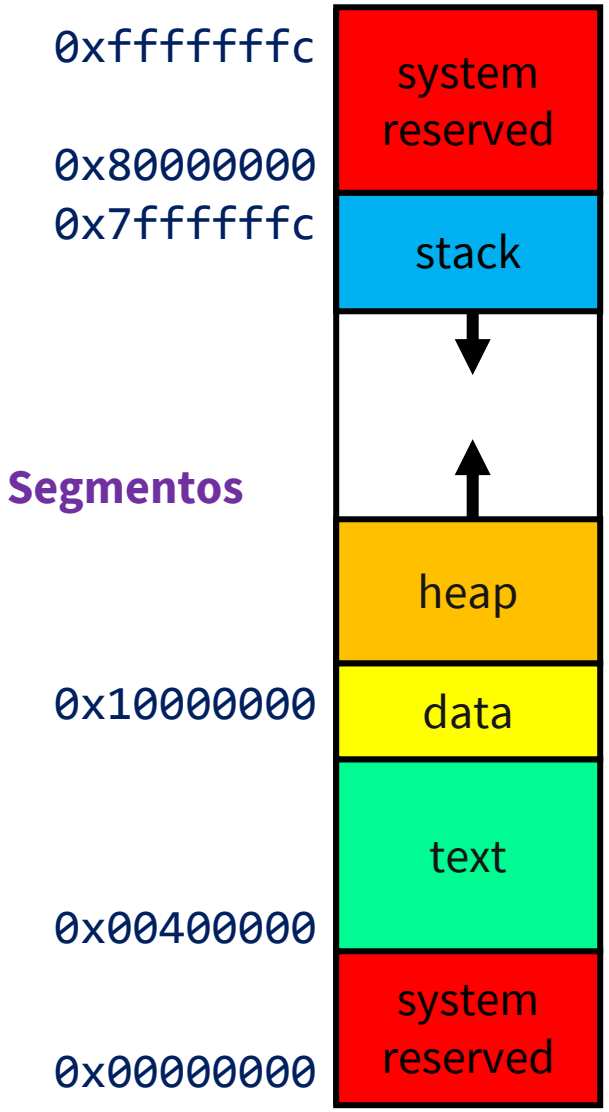
Performance em Sistemas Ciberfísicos

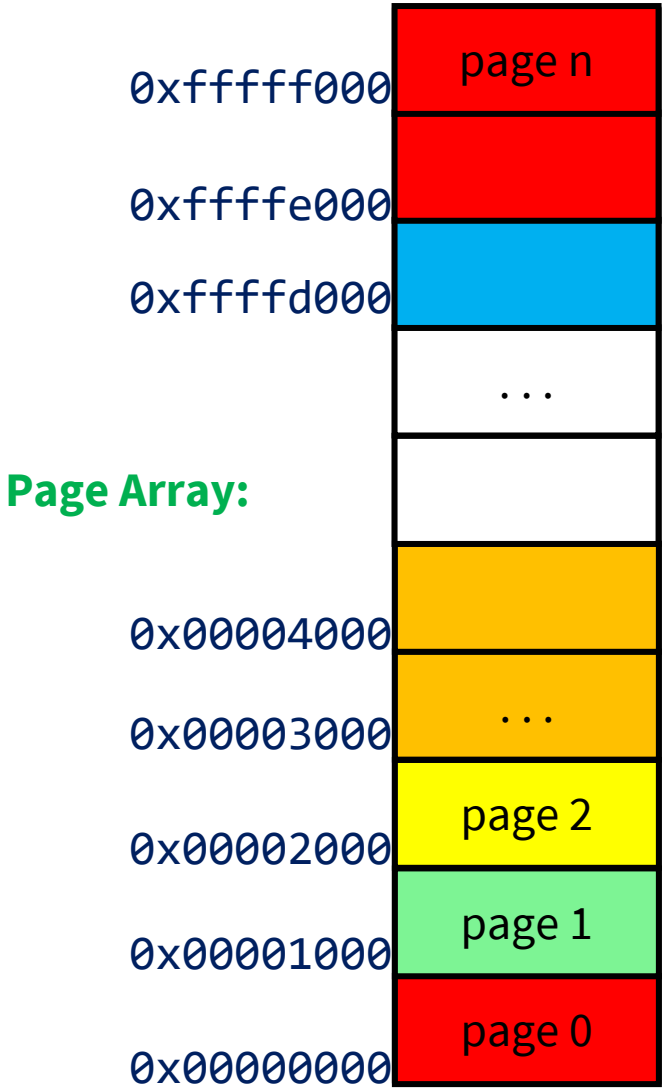
Aula 8 – Processos e Paralelismo

Frank Coelho de Alcantara – 2023 -1

Uma Memória. Muitas faces

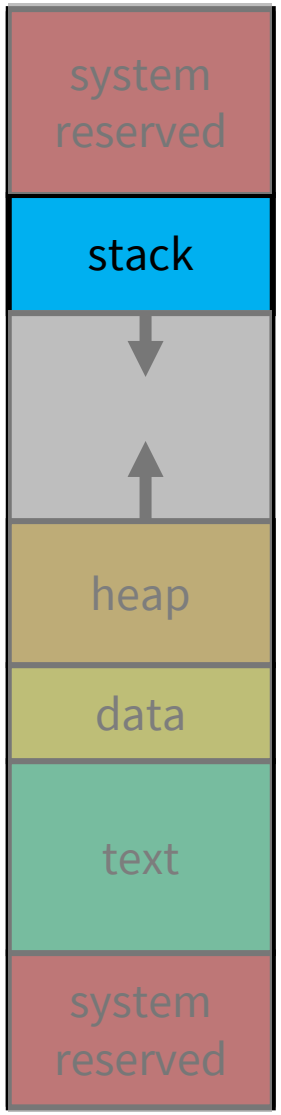






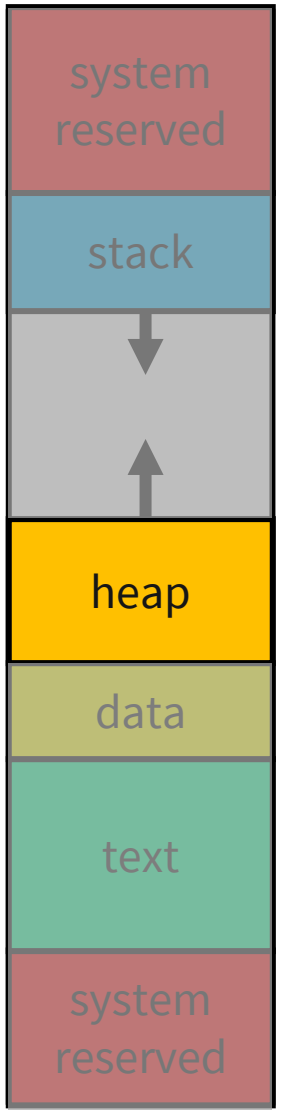
Cada segment usa o mesmo número de páginas

Processos - Stack



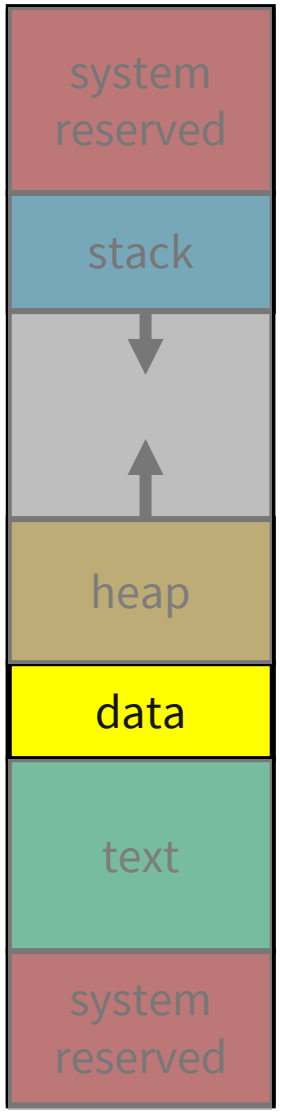
- **Stack (Pilha):** O stack é uma área de memória reservada a um processo específico usada para armazenar variáveis locais e informações de controle de fluxo, como endereços de retorno de funções.
- O *stack* é uma pilha de quadros de pilha (*stack frames*), onde cada quadro corresponde a uma chamada de função. O stack tem um tamanho limitado e é gerenciada pelo compilador.

Processos - Heap



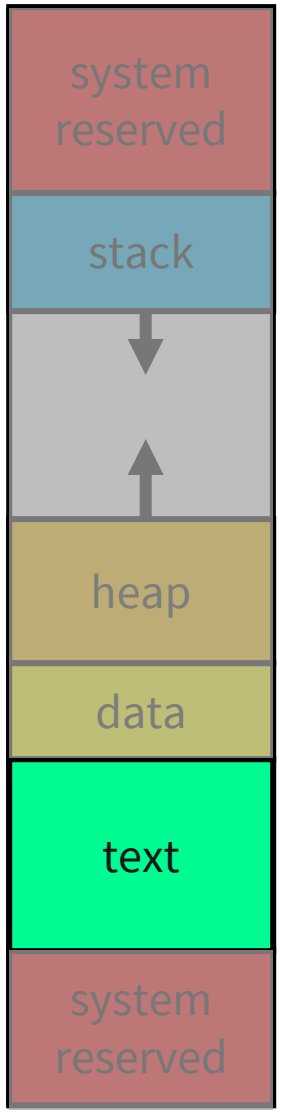
- **Heap (Monte):** O *heap* é uma área de memória de um processo específico usada para alocar memória dinamicamente durante a execução do programa.
- Sempre que um objeto é criado usando o operador *new* em C++, a memória necessária para este objeto é alocada no heap. O tamanho da heap é limitado pela quantidade de memória disponível no sistema.

Processos - Data



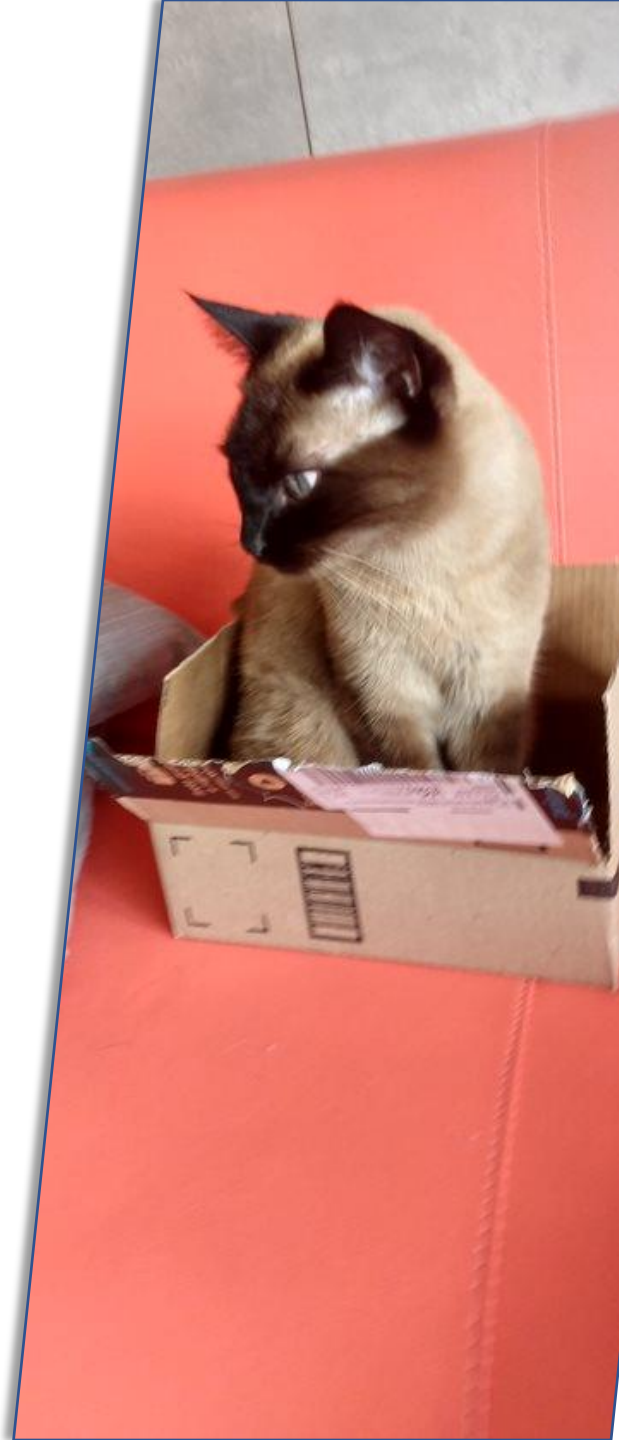
- **Data:** A área de dados é usada para armazenar variáveis globais e estáticas do programa. É dividida em duas subáreas: a área de dados inicializados e a área de dados não inicializados (também chamada de BSS - Block Started by Symbol). A área de dados inicializada contém variáveis globais e estáticas com valores iniciais especificados no código-fonte, enquanto a área de dados não inicializada contém variáveis globais e estáticas não inicializadas ou inicializadas com zero.

Processos - Text



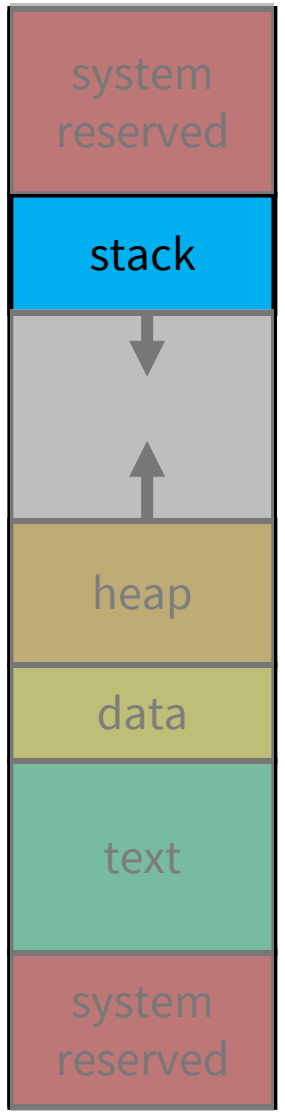
- **Text (Texto):** A área de texto é a parte da memória do processo onde o código executável do programa está armazenado. A área de texto é somente leitura, ou seja, não é possível modificar o código do programa em tempo de execução.

Entendendo o Stack

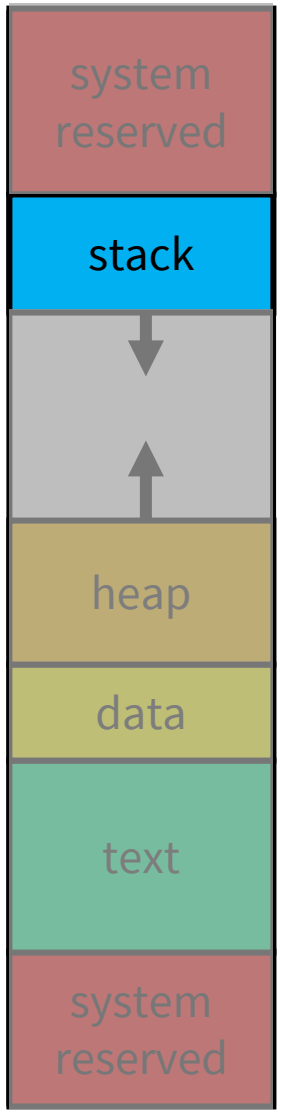


Processos - Stack

- **Stack (Pilha):** O *stack* é uma pilha de quadros de pilha (*stack frames*), onde cada quadro corresponde a uma chamada de função. O stack tem um tamanho limitado e é gerenciada pelo compilador.
- **Stack Frame**: Um stack frame (quadro de pilha, em português) é uma estrutura de dados, uma pilha, que representa uma instância de uma função em execução na pilha de memória.



Processos – Stack Frame – Informações



1. Valores dos argumentos da função: Parâmetros passados para a função.
2. Variáveis locais: Variáveis que são declaradas e utilizadas apenas dentro do escopo da função.
3. Endereço de retorno: O endereço de memória para o qual o controle será retornado após a conclusão da função.
4. Informações sobre o quadro de pilha anterior: O endereço do quadro de pilha anterior (geralmente armazenado no registro base pointer) é usado para restaurar o contexto da função chamadora.

Stack Frame – Entendendo esta Pilha

- A pilha (stack, em inglês) é uma região especial da memória de um programa que funciona como uma estrutura de dados do tipo "último a entrar, primeiro a sair" (LIFO - Last In, First Out). Ela é usada para armazenar informações temporárias, como endereços de retorno de funções, variáveis locais e argumentos de funções.
- A pilha é gerenciada por dois registradores principais: o stack pointer (rsp, em assembly x86-64) e o base pointer (rbp, em assembly x86-64). O stack pointer aponta para o topo da pilha, enquanto o base pointer aponta para a base do quadro de pilha atual.

Stack Frame – Entendendo esta Pilha

- Quando uma função é chamada, o endereço de retorno é armazenado no topo da pilha. A função pode então alocar espaço na pilha para suas variáveis locais e salvar os registradores que serão usados durante a execução da função. Após a função ser executada, os valores dos registradores são restaurados, as variáveis locais são liberadas e o endereço de retorno é recuperado do topo da pilha.
- As operações de "push" e "pop" são usadas para adicionar e remover valores da pilha, respectivamente. Quando um valor é adicionado à pilha (push), o stack pointer é decrementado; quando um valor é removido (pop), o stack pointer é incrementado.

Processos – Stack Frame – Armazena

- **Valores dos argumentos da função**: parâmetros passados para a função.
- **Variáveis locais**: utilizadas apenas dentro do escopo da função.
- **Endereço de retorno**: O endereço de memória para o qual o controle será retornado.
- **Informações sobre o quadro de pilha anterior**: O endereço do quadro de pilha anterior usado para restaurar o contexto da função chamadora.

Endereço de retorno	rsp + 24 (exemplo em x86-64, pode variar)
Argumento: a	rsp + 16
Argumento: b	rsp + 8
Antigo valor de rbp	rsp (rbp)
Variável local: result	rsp - 4

Um Pouco de Assembly – godbolt.org

```
int add(int a, int b) {
    int result = a+b;
    return result;
}
```

Código em C++

Endereços em Hexadecimal

OPCODES Instruções em Hexadecimal

```
add(int, int):
 55
401106      push   rbp
 48 89 e5
401107      mov    rbp,rsp
 89 7d fc
40110a      mov    DWORD PTR [rbp-0x4],edi
 89 75 f8
40110d      mov    DWORD PTR [rbp-0x8],esi
 8b 55 fc
401110      mov    edx,DWORD PTR [rbp-0x4]
 8b 45 f8
401113      mov    eax,DWORD PTR [rbp-0x8]
 01 d0
401116      add    eax,edx
```

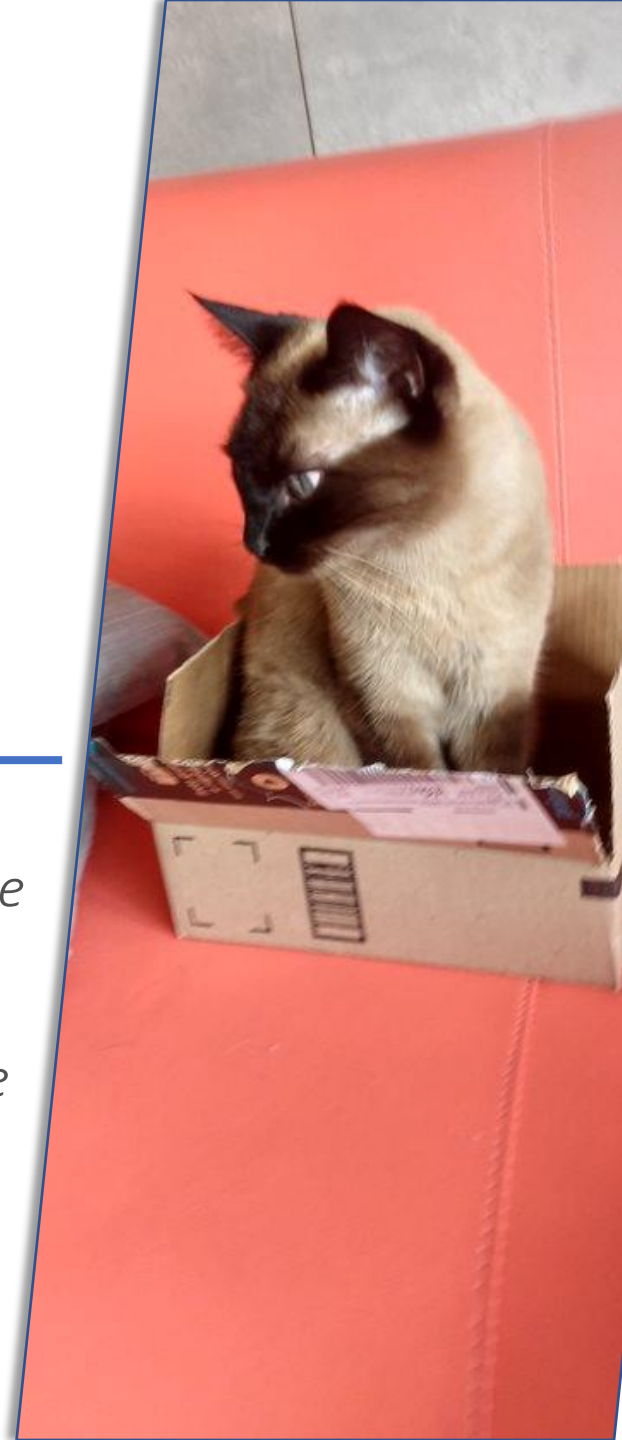
Algumas Instruções em Assembly

- ***push rbp***: Salva o valor do registro base pointer (rbp) na pilha, armazenando o endereço do quadro de pilha anterior.
- ***mov rbp, rsp***: Move o valor do registro stack pointer (rsp) para o registro base pointer (rbp). Isso define o início do novo quadro de pilha.
- ***mov DWORD PTR [rbp-0x4], edi***: Move o valor do registro edi (primeiro argumento da função) para a variável local no endereço [rbp-0x4].
- ***mov DWORD PTR [rbp-0x8], esi***: Move o valor do registro esi (segundo argumento da função) para a variável local no endereço [rbp-0x8].
- ***mov edx, DWORD PTR [rbp-0x4]***: Move o valor da variável local no endereço [rbp-0x4] para o registro edx.

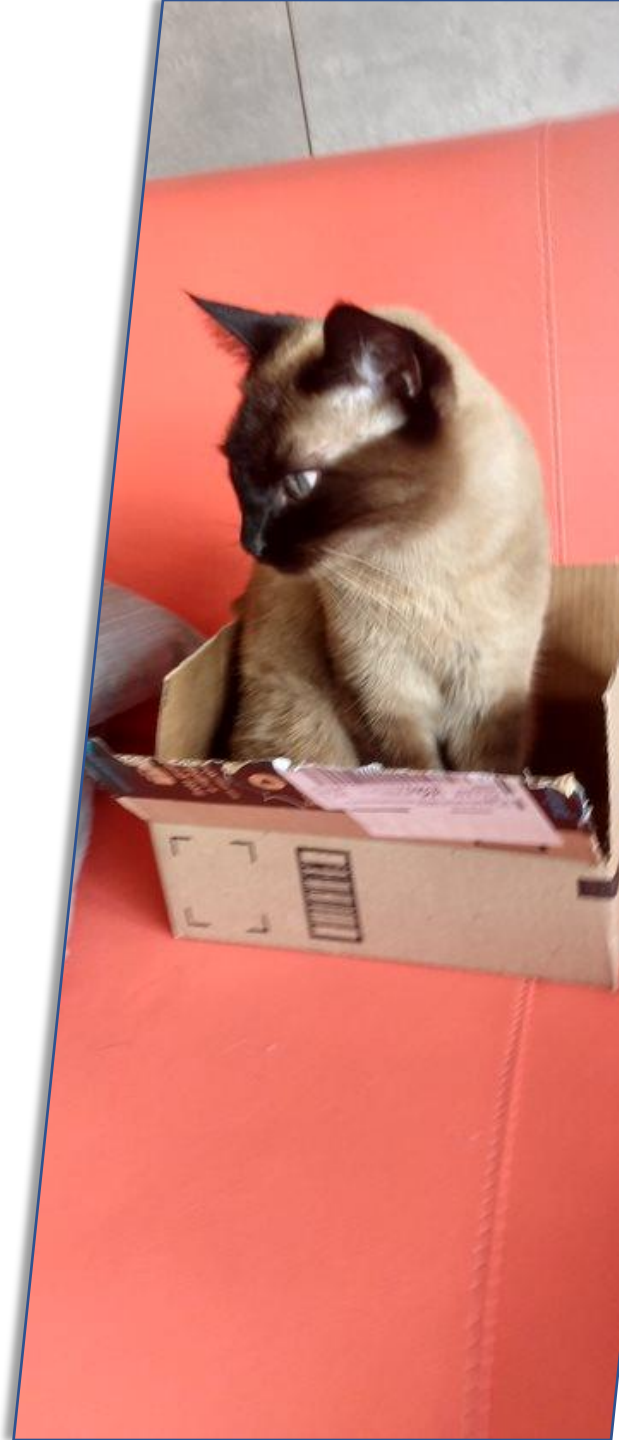
Atividade Prática em Python

Você tem 20 minutos para fazer um programa em Python, no Godbolt.org, com duas operações aritmética e no mínimo um laço for para tentar entender o Assembly gerado por dois compiladores diferentes.
Aproveite para pesquisar se não existem otimizações que você pode fazer no compilador.

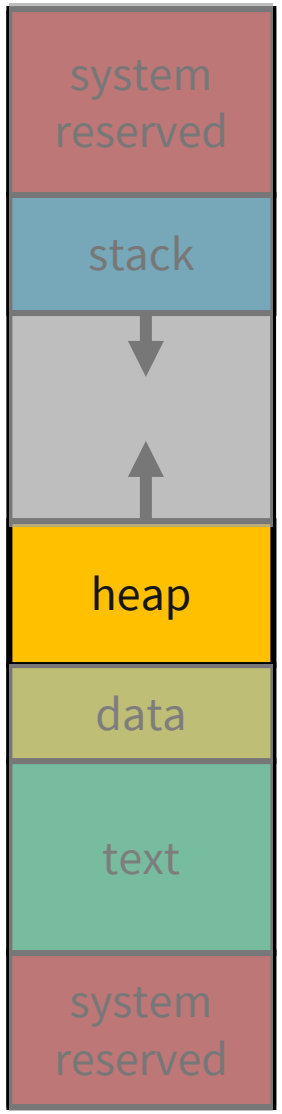
Não! Esta atividade não vale ponto!



Entendendo o Heap

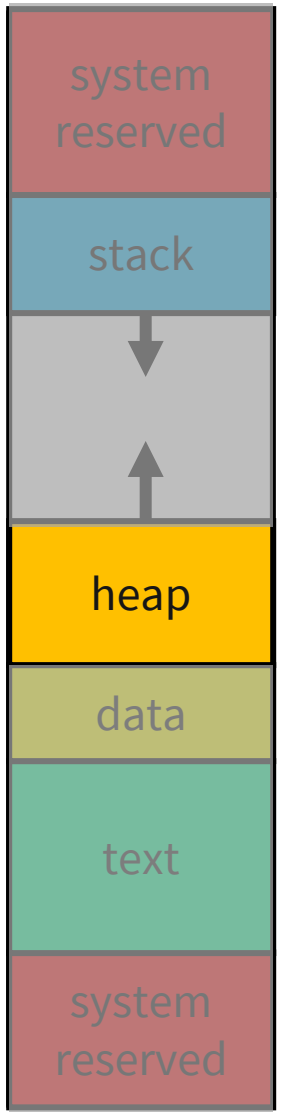


Processos – Heap



- **Heap**: uma área da memória do processo gerenciada dinamicamente, utilizada para alocar e desalocar memória conforme seja necessário durante a execução de um programa.
- Diferentemente do stack (pilha), que segue uma estrutura de dados do tipo LIFO (*last-in, first-out*) e tem um tamanho fixo, o *heap* permite a alocação de blocos de memória de tamanho variável e não segue uma ordem específica.

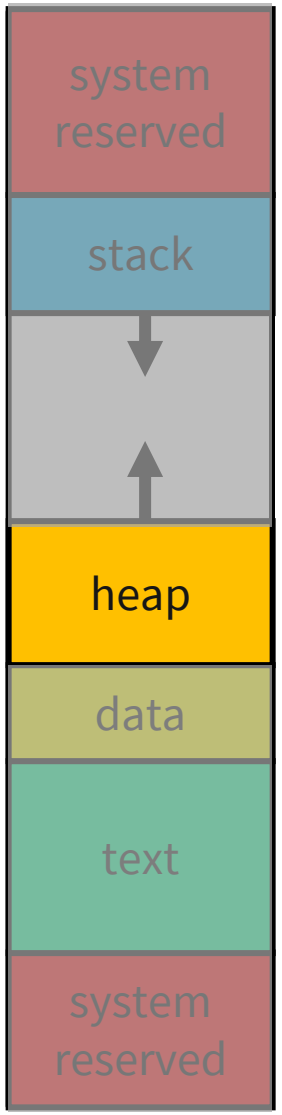
Processos – Heap – Vantagens e Desvantagens



- **Vantagens do heap:**
  - **Alocação dinâmica de memória**: Permite alocar e desalocar memória durante a execução do programa, adaptando-se às necessidades de memória.
  - **Tempo de vida mais longo**: Os objetos alocados no heap têm um tempo de vida maior que o escopo ou a função em que foram criados, permitindo que os dados persistam até serem explicitamente liberados.

Processos – Heap – Vantagens e Desvantagens

- **Desvantagens do heap**:
  - **Gerenciamento de memória: O desenvolvedor deve garantir que a mem**ória alocada seja liberada apropriadamente, caso contrário, isso pode levar a vazamentos de memória.
  - **Fragmentação**: A alocação e desalocação frequente de blocos de memória pode resultar em fragmentação do heap, afetando a eficiência da alocação de memória.
  - **Velocidade**: A alocação e liberação de memória no heap é geralmente mais lenta do que na stack, devido ao gerenciamento de memória mais complexo.



Processos – Heap – Um Pouco de Código

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> *my_vector = new std::vector<int>(10);

    // Uso do vetor
    (*my_vector)[0] = 42;

    // Liberação da memória
    delete my_vector;

    return 0;
}
```

Processos – Heap – Um Pouco de Código

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> *my_vector = new std::vector<int>(10);

    // Uso do vetor
    (*my_vector)[0] = 42;

    // Liberação da memória
    delete my_vector;

    return 0;
}
```

- O sistema verifica se há espaço suficiente disponível no heap para alocar o objeto solicitado.
- Se houver espaço suficiente, o sistema aloca a memória necessária e retorna um ponteiro para o bloco de memória alocado.

Processos – Heap – Um Pouco de Código

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> *my_vector = new std::vector<int>(10);

    // Uso do vetor
    (*my_vector)[0] = 42;

    // Liberação da memória
    delete my_vector;

    return 0;
}
```

- Se não houver espaço suficiente, o sistema lança uma exceção do tipo ***std::bad_alloc***.
- Após usar o vetor, é importante liberar a memória alocada usando o operador 'delete'. Isso informa ao sistema que pode ser devolvida ao heap.

Processos – Heap – Um Pouco de Código

- Ao chamar o operador new, o gerenciador de memória:
  1. Consulta a **tabela de alocações** (*heap metadata*) para encontrar um bloco de memória livre que seja suficientemente grande para armazenar o objeto std::vector<int> com capacidade para 10 elementos inteiros.
  2. Atualiza a tabela de alocações para marcar o bloco de memória como "ocupado" e registrar informações sobre a alocação.
  3. Retorna um ponteiro para o bloco de memória alocado.

Processos – Heap – A Tabela de Alocação (Meta Dados)

- A tabela de metadados do heap é uma estrutura de dados interna usada pelo gerenciador de memória para manter o controle de blocos de memória alocados e livres no heap. A tabela de metadados do heap pode ser implementada usando várias estruturas de dados, como listas ligadas, árvores binárias, bitmaps, entre outras.
- A escolha da estrutura de dados depende das necessidades de desempenho e eficiência da implementação específica do gerenciador de memória.

Processos – Heap – A Tabela de Alocação Armazena

1. **Endereço do bloco de memória**: o endereço base do bloco de memória no heap.
2. **Tamanho do bloco**: o tamanho do bloco de memória em bytes.
3. **Estado do bloco**: se o bloco de memória está alocado (em uso) ou livre (disponível para alocação).
4. **Informações adicionais**: a tabela de metadados do heap também pode incluir informações adicionais, como ponteiros para blocos adjacentes, tamanho do bloco de memória solicitado pelo usuário (antes de adicionar quaisquer sobrecargas de gerenciamento de memória), etc.

Processos – Heap – A Tabela de Alocação - Windows

1. A tabela de alocações do heap no Windows é implementada como uma árvore **binária balanceada (RB-Tree)** de blocos de memória alocados, onde cada nó da árvore representa um bloco de memória contém informações sobre o bloco de memória, como seu tamanho, endereço base e estado (alocado ou livre).
2. Além da árvore RB-Tree, o Heap Manager também mantém outras estruturas de dados internas, como listas de blocos livres e blocos grandes, para ajudar no gerenciamento eficiente da memória.

Processos – Heap – A Tabela de Alocação - Windows

3. A tabela de alocações do heap é armazenada no próprio heap, em um bloco especial chamado Segmento do Heap. O segmento do heap é uma região contígua de memória que contém uma lista blocos de memória alocados e livres, Além da Tabela de Alocações do heap.
4. A tabela de alocações do heap é protegida contra leitura/gravação não autorizada por meio de mecanismos de proteção de memória fornecidos pelo sistema operacional. OHeap Manager pode usar a API do sistema operacional para marcar o segmento do heap como protegido.

Processos – Segmentos e Mais Segmentos

1. **Segmentos de heap**: são os segmentos de memória principais usados pelo Heap Manager para armazenar os blocos de memória alocados pelo processo. Cada segmento de heap é um bloco de memória contíguo, com um tamanho inicial definido pelo Heap Manager. Conforme o processo aloca mais memória, o Heap Manager pode expandir o segmento do heap existente ou alocar novos segmentos de heap.

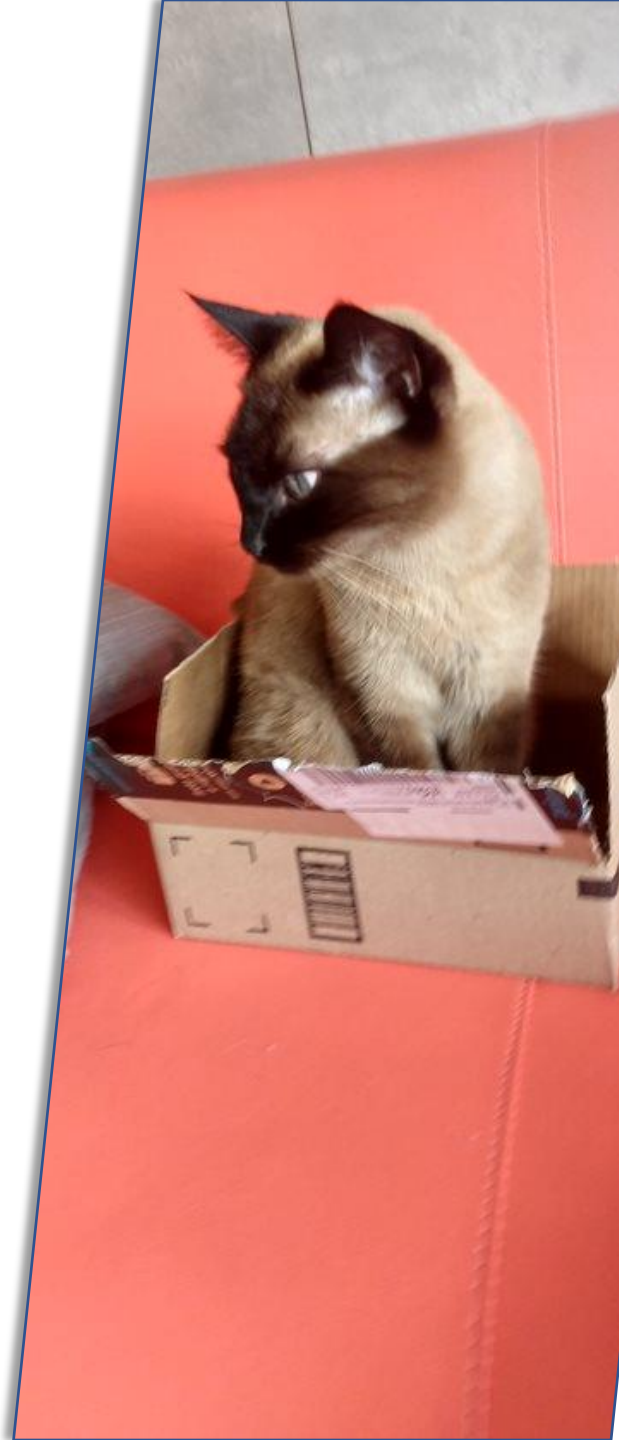
Processos – Segmentos e Mais Segmentos

2. **Segmentos de heap grandes**: são segmentos de memória maiores que um determinado tamanho, definido pelo Heap Manager. Os segmentos de heap grandes são usados para armazenar blocos de memória maiores que o tamanho máximo permitido em um segmento de heap padrão. Os segmentos de heap grandes são gerenciados separadamente do heap principal, mas ainda fazem parte do heap global do processo.

Processos – Segmentos e Mais Segmentos

3. **Segmentos de heap local**: são segmentos de memória adicionais usados pelo Heap Manager para armazenar os dados de configuração do heap e as tabelas de alocação de memória. Os segmentos de heap local não são usados para armazenar os blocos de memória alocados pelo processo.
4. **Segmentos de heap de pilha**: são segmentos de memória usados pelo Heap Manager para armazenar as pilhas dos threads do processo. Cada thread tem seu próprio segmento de heap de pilha, que é usado para armazenar as chamadas de função e as variáveis locais da thread.

Paralelismo



Voltando a Performance

- Em Computação a Performance e definida por:
  - Necessidades do Algoritmo (o que será feito?)
  - Recursos Computacionais (quanto isso custa?)
- Performance é uma medida de tão bem as necessidades do algoritmo podem ser satisfeitas pelos recursos computacionais.
- As medidas de performance refletem a qualidade das decisões que tomamos na implementação do algoritmo.

Performance é um Problema Antigo

“The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”

Charles Babbage, 1791 – 1871

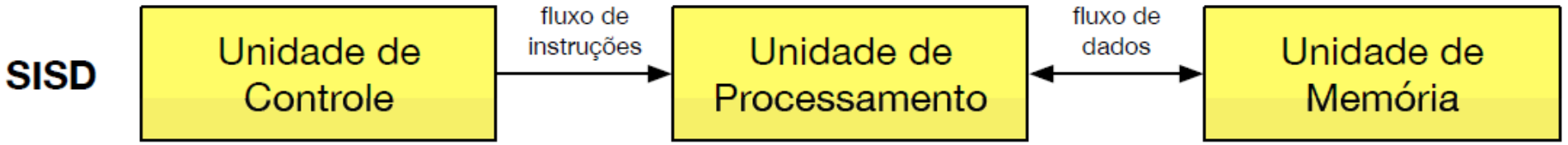
Porque Computação em Paralelo

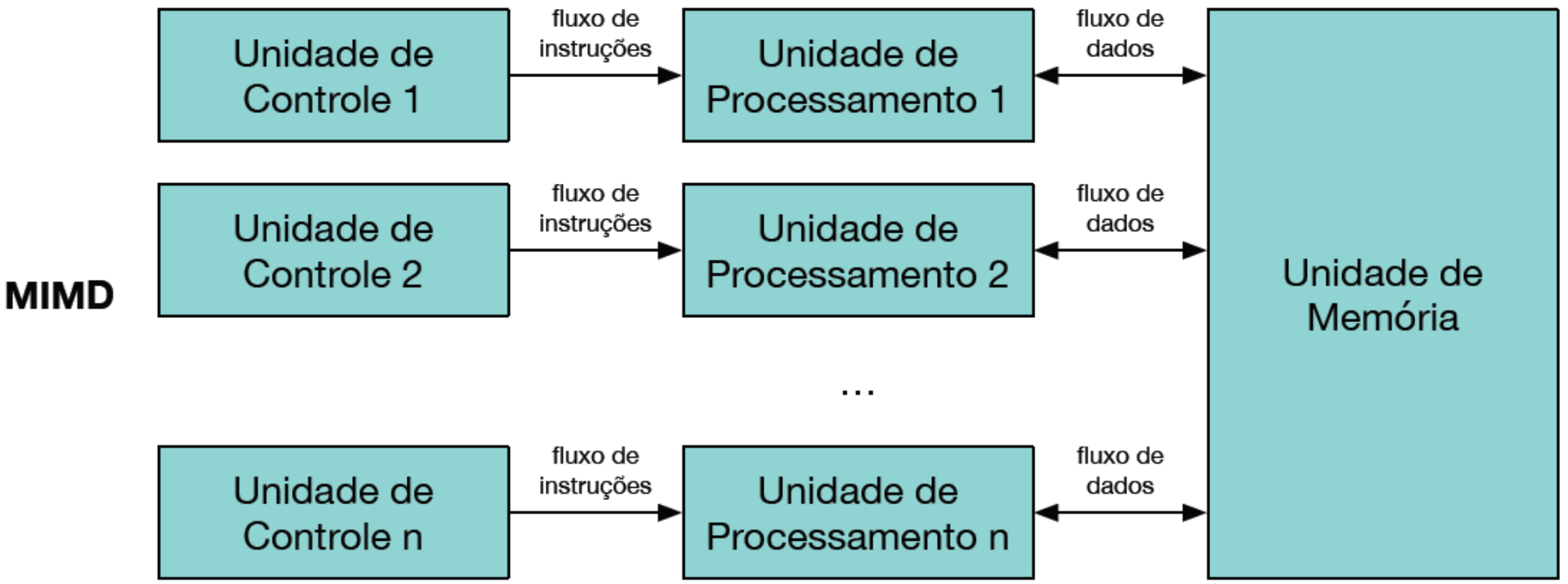
- **Paralelismo**: realizar várias tarefas na mesma unidade de tempo;
- **Objetivo Principal**: melhorar a performance, tempo de execução e throughput (quantidade de tarefas por unidade de tempo);
- **Objetivos Secundários**: reduzir o consumo de energia; reduzir a complexidade; diminuir os custos; melhorar a escalabilidade.

Taxonomia de Flynn

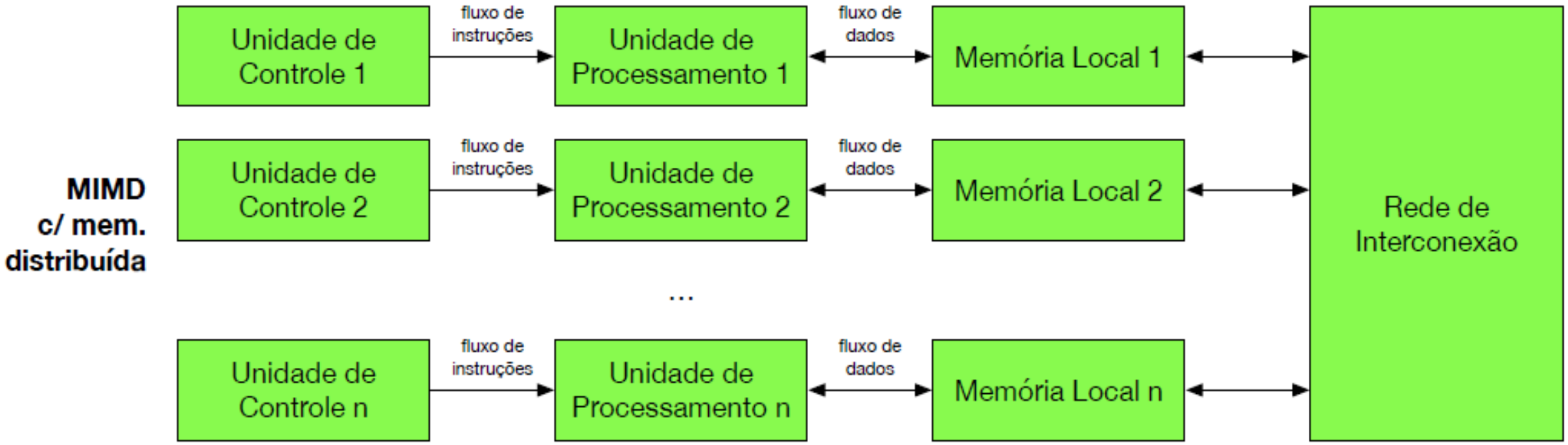
- Mike Flynn, "*Very High-Speed Computing Systems*", Proc. de IEEE, 1966
- **SISD**: Single instruction operates on single data element;
- **SIMD**: Single instruction operates on multiple data elements (vetores e arrays);
- **MISD**: Multiple instructions operate on single data element (processadores de *streams*)
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)

Arquiteturas





Arquiteturas



Tipos de Paralelismo – Instruction Level Parallelism

- Instruction Level Parallelism (ILP)
  - Instruções diferentes da mesma pilha podem ser executadas em paralelo;
  - Pipelining, *out-of-order execution*, *speculative execution*, VLIW (*Very Long Instruction Word*);
  - Dataflow (fluxo de dados em uma rede de processadores).

Tipos de Paralelismo – Data e Task Level

- Data Parallelism
  - Dados diferentes operados em paralelo;
  - SIMD: processamento de vetores e arrays;
  - Systolic arrays, streaming processors
- Task Level Parallelism
  - tasks/threads podem ser executados em paralelo
  - Multithreading
  - Multiprocessing (multi-core)

Streaming Processors

Os processadores de streaming são classificados como uma variante da taxionomia SIMD, projetados para executar operações simultâneas em um grande conjunto de dados.

Nessa arquitetura, várias unidades de processamento executam a mesma instrução em diferentes conjuntos de dados ao mesmo tempo, o que é adequado para tarefas que envolvem operações simples e repetitivas, como processamento de áudio e vídeo em tempo real.

Os processadores de streaming são um tipo especializado de processador SIMD, com foco específico em fluxos contínuos *(streaming)* de dados em vez de matrizes de dados estáticos.

Streaming Processors

1. **AMD Radeon**: as GPUs da AMD Radeon usam uma arquitetura baseada em streaming processors, otimizada para processar fluxos de dados em paralelo para jogos, gráficos e aplicações de computação intensiva.
2. **NVIDIA Tesla**: as GPUs NVIDIA Tesla são projetadas para computação científica, análise de dados e *machine learning*. Elas usam uma arquitetura de *streaming* processor para processar grandes quantidades de dados em paralelo, permitindo a execução de tarefas complexas de forma rápida e eficiente.
3. **Intel Xeon Phi**: o co-processador Intel Xeon Phi é projetado para computação de alta performance, especialmente em aplicações de processamento paralelo intensivo. Ele usa uma arquitetura de *streaming* para executar operações matemáticas em paralelo, para tarefas de computação intensiva.

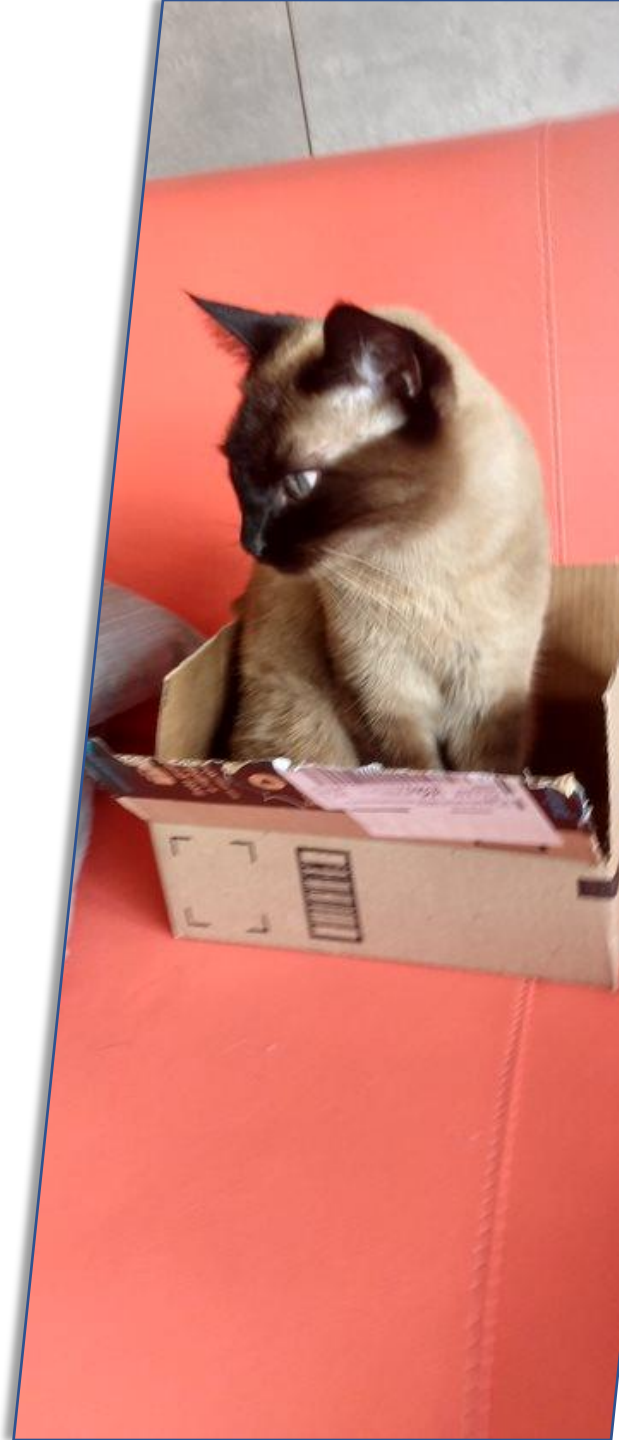
Systolic Array

- Em uma arquitetura *Systolic Array*, os elementos de processamento são organizados em uma rede de células de. Otimizadas para executar operações matemáticas específicas, como multiplicação e soma, em um fluxo contínuo de dados. Cada célula de processamento realiza uma pequena parte do cálculo, e os dados são transmitidos entre as células em uma ordem predefinida.
- As *Systolic Arrays* são usadas em aplicações que exigem processamento de sinais em tempo real, como processamento de imagem e vídeo, processamento de fala e processamento de radar e sonar.

Systolic Array - Exemplos

- **Google TPU (Tensor Processing Unit)**: um processador de tensor projetado para acelerar operações de aprendizado de máquina e inteligência artificial. Ele usa uma arquitetura de *Systolic Array* para processar grandes matrizes de dados em paralelo, permitindo a execução rápida de tarefas de aprendizado de máquina.
- **Xilinx FPGAs**: As FPGAs (*Field-Programmable Gate Arrays*) da Xilinx são usadas em aplicações de processamento de sinal digital, processamento de imagem e inteligência artificial.

Task Level Parallelism



Task Level Parallelism

- Dividir um problema em um conjunto de tarefas relacionadas:
- De forma explícita: Programação em Paralelo
  - Simples quando as tarefas são claras;
  - Complexo quando a divisão em tarefas não é clara.
- De forma implícita: Paralelismo a nível de *thread*
- Também podemos rodar múltiplos processos relacionados a uma tarefa específica. Não existe ganho em cada tarefa.

Ganho de Performance

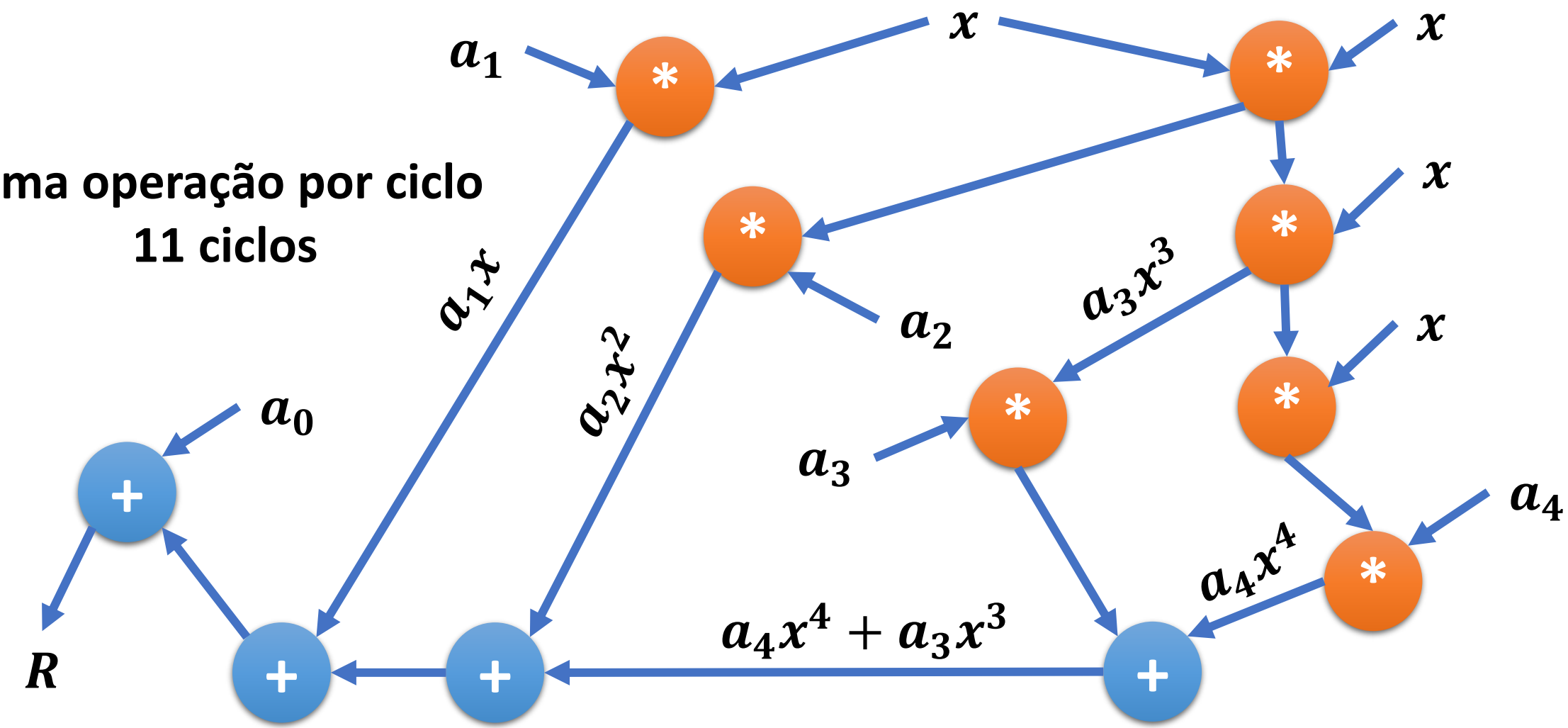
- Considere $f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$
- Considere também que cada operação utiliza 1 ciclo de *clock,* sem custos de comunicação e que cada operação pode ser executada em um processador diferente.
- O quão rápido esta operação será em um processador singular? Assuma que não existe uma *pipeline* ou qualquer outro tipo de execução concorrente.
- O quão rápido será em quatro processadores?

Um processador

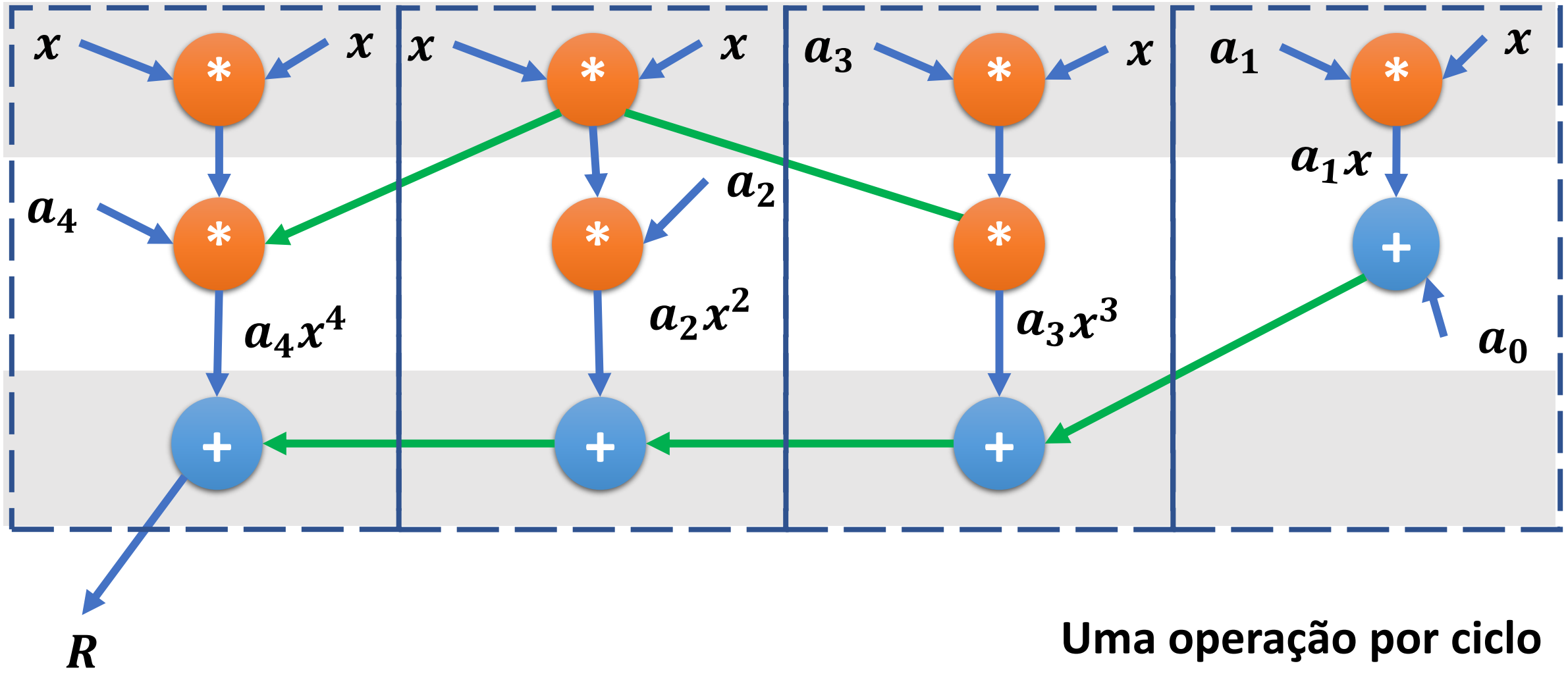
$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Uma operação por ciclo

11 ciclos



Quatro processadores



$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Uma operação por ciclo

3 ciclos

Ganho de Velocidade (*speedup*)

$$Ganho = \frac{11}{3} = 3{,}67$$

Talvez possamos fazer melhor: Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.

$$\boldsymbol{f(x) = (((a_4 x + a_3)x + a_2)x + a_1)x + a_0}$$

Lei de Amdahl

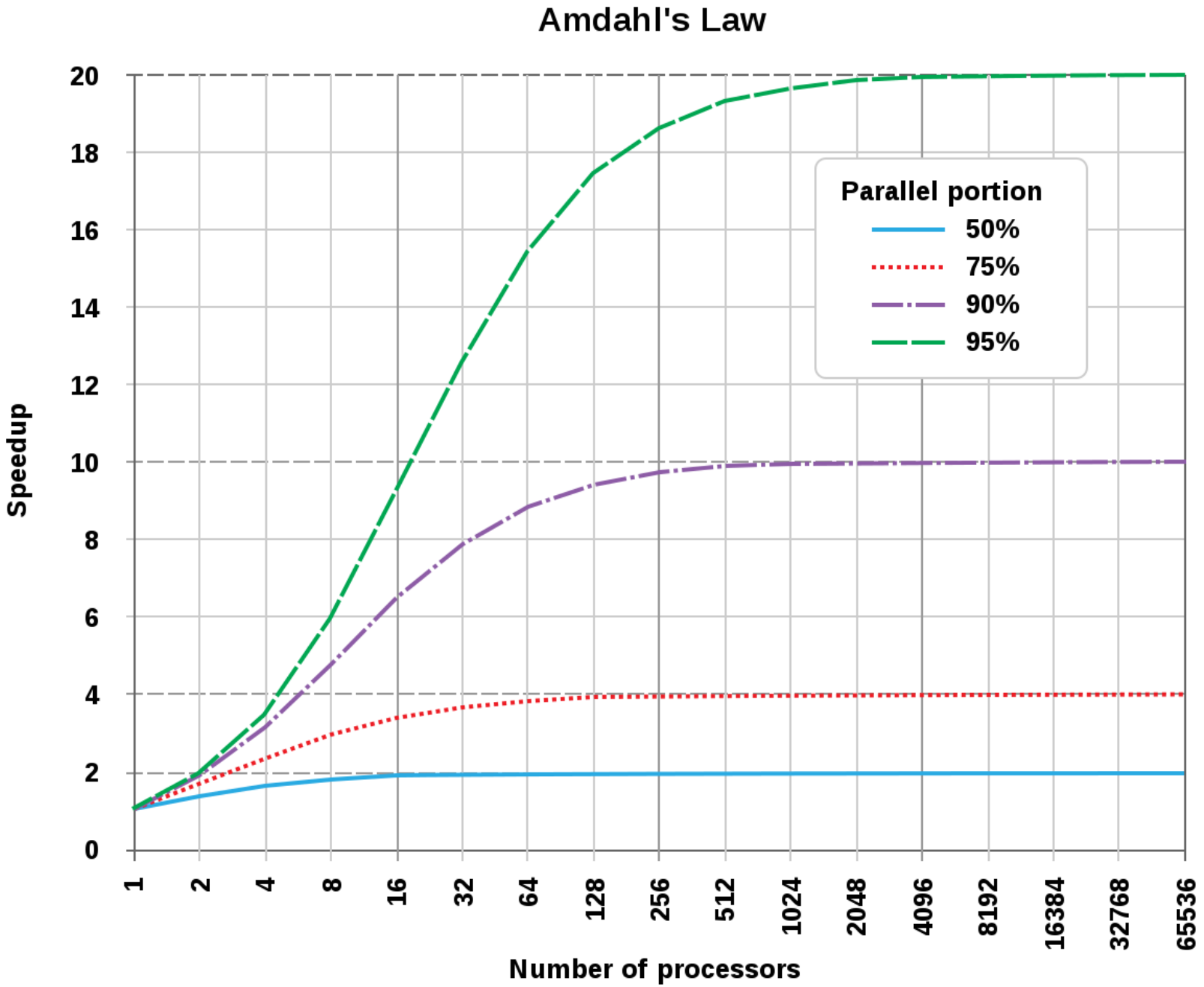
$$Ganho = \frac{Tempo\ em\ um\ processador}{Tempo\ em\ Paralelo} = \frac{1}{\frac{\alpha}{p} + (1 - \alpha)}$$

Onde α representa o tempo das tarefas que não podem ser paralelizadas e p o número de processadores

$$\lim_{p \to \infty} Ganho = \frac{1}{1 - \alpha}$$

Amdahl, "*Validity of the single processor approach to achieving large scale computing capabilities*," AFIPS 1967.

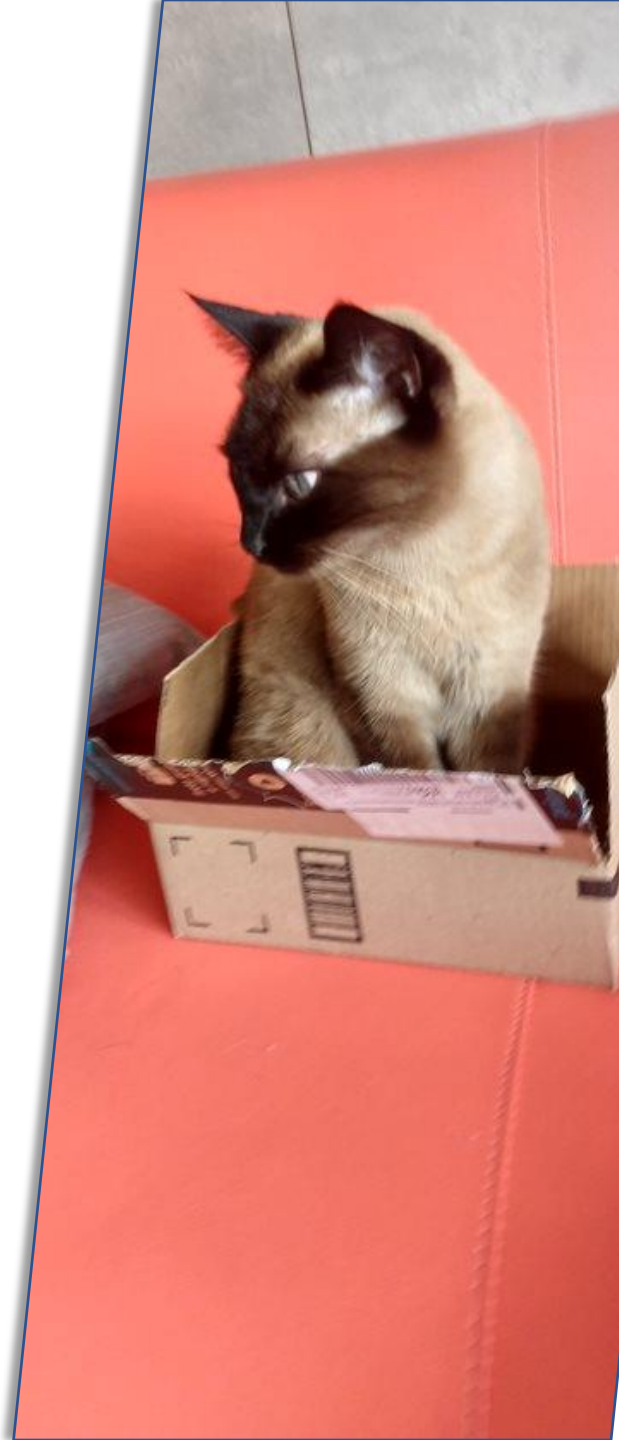
Amdhal Law



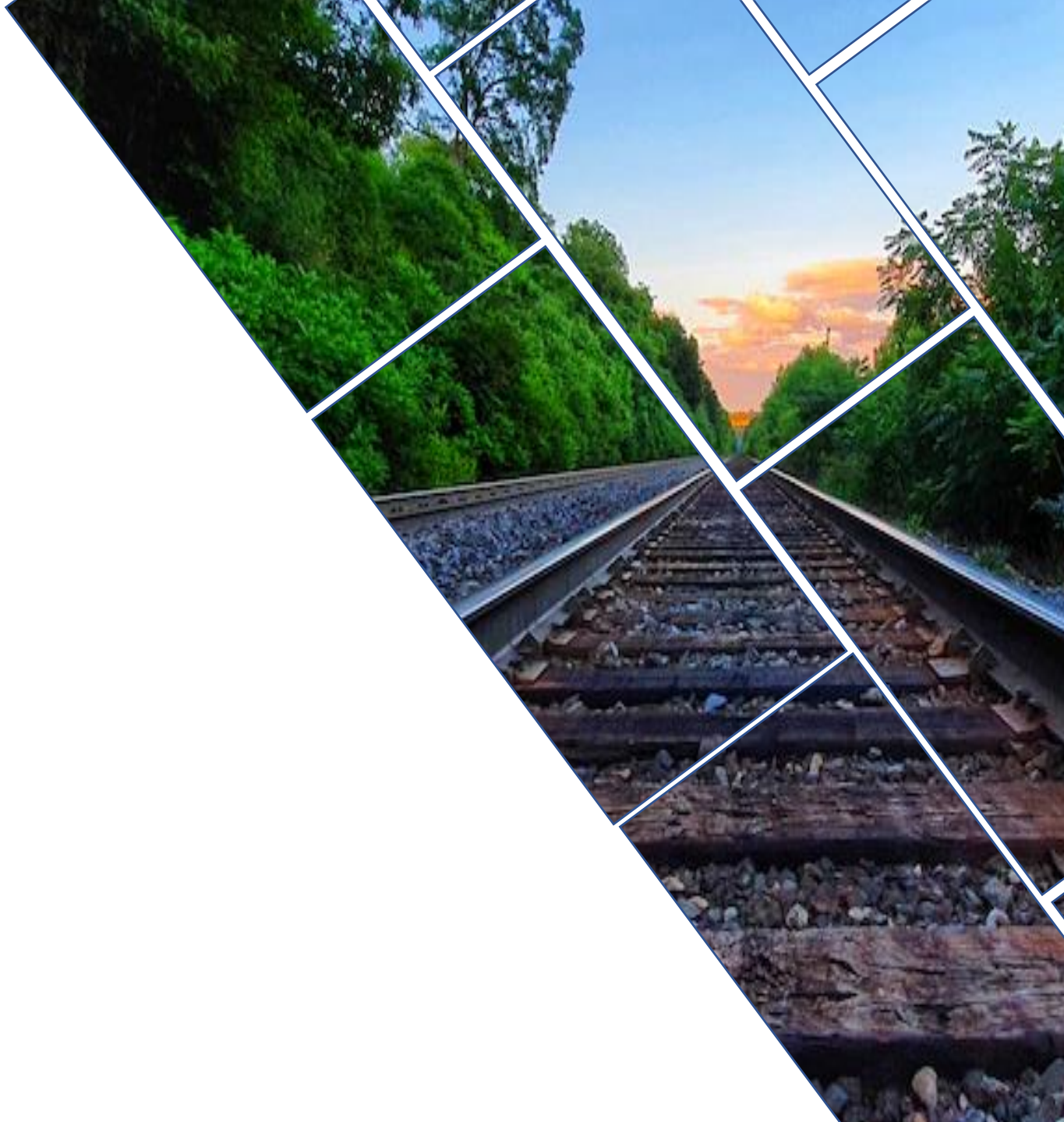
Wikimedia Commons contributors, 'File:AmdahlsLaw.svg', *Wikimedia Commons,* 15 September 2020, 16:23 UTC, <https://commons.wikimedia.org/w/index.php?title=File:AmdahlsLaw.svg&oldid=460310372> [accessed 9 May 2023]

Amdhal Law

Seu trabalho será explicar como é possível chegar a Lei de Amdhal. Para isso, você deve resumir e refazer a matemática apresentada por Amdhal. Não! Esta atividade não vale ponto!.



Obrigado!



Frank Coelho de Alcantara – 2023 -1