



# SPD

## Site com o conteúdo

[https://sites.google.com/ifsudestemg.edu.br/spd/aulas/1-introdução?  
authuser=0](https://sites.google.com/ifsudestemg.edu.br/spd/aulas/1-introdução?authuser=0)

### ▼ Introdução

#### O que são sistemas distribuídos?

São uma coleção de computadores independentes que aparentam a seus usuários ser um sistema único e coerente.

#### Quais são os objetivos de um sistema distribuído?

- Fazer um link entre os usuários e os recursos:
  - Compartilhamento de recursos.
  - Segurança.
  - Reducir a comunicação indesejada.
- Transparência (Iludir o usuário)
- Flexibilidade (É possível adicionar coisas sem que o usuário saiba)
- Escalabilidade (Pode aumentar e diminuir facilmente)

#### Características dos algoritmos descentralizados

- Nenhuma máquina tem informação completa do estado do sistema.
  - Se sai do ar, o sistema continua funcionando.
- As máquinas tomam decisões com informações locais.
- A falha no algoritmo não “quebra” todo o sistema.
- Não existe uma suposição de um relógio global.

# Técnicas de escalabilidade

## Aumentar a disponibilidade de recursos

Assim você deixa os dados serem processados na máquina do cliente para poupar o servidor.

## Balanceamento de carga

Distribuir o sistema por zonas, assim como funciona no sistema DNS que vimos em redes.

## Ocultar a latência de comunicação caso haja grande dispersão geográfica

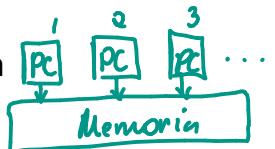
---

## Tipos de Sistemas Distribuídos

### **Computação em Cluster (Cluster Computing)**

Características:

- Possui um ambiente mais homogêneo.
- Hardware: conjunto de PCs ou estações de trabalho semelhantes
- Conexão entre os hardwares: rede local (LAN).
- Software:
  - É normal ter o mesmo SO entre as máquinas.
  - Geralmente, um único programa executado em paralelo.
- Normalmente usada para computação paralela.
- Forte acoplamento entre os nós (Pode ter uma única memória compartilhada por vários computadores diferentes).



Funcionamento:

Aqui temos um nó mestre que distribui as tarefas para as máquinas.  
Temos até uma migração de processos.

### **Computação em Grade (Grid Computing)**

Características:

## Heterogeneidade

- Hardware de diferentes organizações são reunidos e dispersos entre elas. (Conjunto de empresas colaboram)
- Para permitir a colaboração de um grupo de pessoas ou instituições.
- É uma rede de pesquisa mundial para criar novos serviços de rede.
- A RNP faz parte dela. → Planetlab

## Computação em Nuvem (Cloud Computing)

Características:

↳ organização aberta como o gmail

- Quase igual a em grade.
- Uso de um recurso de computação:
  - Como uma máquina virtual (VM).
  - Um armazenamento.
  - Uma aplicação.
- Igual ao consumo da eletricidade ("serv. terceirizado")
  - Ao invés de ter que construir e manter infraestruturas de computação em casa ou na empresa.

## Sistemas de Informação Distribuídos

Sistemas corporativos para integrar diversas aplicações onde a interoperabilidade se mostrou penosa.

- Sistemas de processamento de Transações.
- Integração de aplicações empresariais.

Processamento de transações:

Requer primitivas especiais, que devem ser fornecidas pelo sistema distribuído ou pela linguagem.

Características - ACID:

- Atômicas: para o mundo exterior, indivisível.
- Consistentes: não viola as invariantes do sistema(dados válidos antes e depois da transação de transferência).
- Isolada: transações concorrentes não interferem uma com as outras.

- Duráveis: uma vez comprometida uma transação, as alterações são permanentes.

Pode ter transações aninhadas.

## Sistemas Distribuídos Pervasivos

- Instabilidade é o comportamento esperado desses sistemas.
- Disponibilidade de computação móveis e embarcados pequenos.
  - Alimentação por bateria.
  - Mobilidade.
  - Conexão sem fio.

## Qual a diferença entre paralelização e distribuição

Uma das possibilidades é que a paralelização está associada a memória compartilhada e a distribuição a troca de mensagens.

---

## Tipos de transparência

- **Transparência de Acesso:** Esconde do usuário as diferenças na representação dos dados e na forma como os recursos são acessados. Isso permite que o usuário acesse recursos distribuídos da mesma maneira que acessaria recursos locais, sem estar ciente da localização ou distribuição dos recursos.
- **Transparência de Localização:** Oculta do usuário onde exatamente um recurso está localizado fisicamente. Isso permite que os recursos sejam movidos de um local para outro sem que o usuário perceba, facilitando a reorganização e a manutenção do sistema.
- **Transparência de Migração:** Esconde do usuário o fato de que um recurso pode ser movido de um local para outro enquanto está em uso. Isso permite que recursos sejam transferidos entre diferentes sistemas ou locais sem interrupção para os usuários que estão utilizando esses recursos.
- **Transparência de Replicação:** Oculta do usuário o fato de que existem múltiplas cópias de um recurso distribuído em diferentes locais. Isso permite que o sistema distribuído forneça redundância e tolerância a

falhas, garantindo que os recursos estejam disponíveis mesmo em caso de falha em um dos locais.

- **Transparência de Concorrência:** Esconde do usuário as complexidades relacionadas à concorrência e à coordenação entre os diferentes processos que compartilham recursos distribuídos. Isso permite que múltiplos usuários acessem e modifiquem os mesmos recursos de forma concorrente sem que isso resulte em inconsistências ou conflitos.
- **Transparência de Falhas:** Oculta do usuário as falhas que ocorrem no sistema e os mecanismos internos de recuperação de falhas. Isso permite que o sistema distribuído continue funcionando de maneira confiável, mesmo em caso de falhas em componentes individuais.
- **Transparência de Realocação:** Esconde do usuário o fato de que um recurso pode ser movido de um dispositivo ou localização para outro durante sua utilização. Isso permite que os usuários acessem e utilizem recursos de maneira transparente, independentemente do dispositivo ou localização em que se encontram.

## ▼ Exercícios

(Pos2015) Um dos objetivos do projeto de um Sistema Distribuído é fornecer transparência, ocultando aspectos distribuídos dos usuários do sistema. Um sistema transparente proporciona um ambiente em que os seus componentes apresentam-se logicamente centralizados, mesmo fisicamente separados. Entre os vários tipos de transparência que os sistemas distribuídos podem fornecer, o ocultamento do fato de que há várias cópias de um recurso disponíveis no sistema é conhecido como  
(A) transparência de acesso.  
(B) transparência de transação.

(C) transparência de replicação.

(D) transparência de concorrência

(E) transparência de migração.

Resposta:

A transparência de replicação é o conceito que oculta do usuário do sistema o fato de que existem várias cópias de um recurso disponíveis no sistema. Isso significa que, para o usuário, o acesso ao recurso parece ser único, mesmo que haja múltiplas cópias distribuídas pelo sistema. Essa transparência é importante para garantir que o usuário não

precise se preocupar com a localização ou a disponibilidade do recurso replicado, simplificando assim o acesso e uso do sistema distribuído.

---

Em sistemas de arquivos distribuídos, o requisito no qual os programas clientes não devem conhecer a distribuição de arquivos, sendo que um único conjunto de operações é fornecido para acesso a arquivos locais e remotos, é denominado transparência de

- A) acesso.
- B) desempenho. ✓
- C) escala. ✓
- D) localização.
- E) mobilidade. ✓

Resposta:

Neste contexto, a transparência de acesso refere-se à capacidade de ocultar do usuário a distinção entre arquivos locais e remotos, fornecendo um único conjunto de operações para acessar arquivos, independentemente de sua localização. Isso significa que os programas clientes podem acessar arquivos da mesma maneira, independentemente de estarem armazenados localmente ou em um local remoto, sem precisar estar cientes da distribuição física dos arquivos. Isso simplifica o desenvolvimento de aplicativos e melhora a usabilidade do sistema de arquivos distribuídos.

---

(2010, SEPLAG-DF, analista de sistemas) Sistemas distribuídos são aqueles dispostos em computadores distintos e autônomos, que trabalham juntos para dar a impressão de ser um único sistema, com acesso local a seus usuários. Uma característica importante nesses sistemas é a transparência. Acerca das formas de transparência de um sistema distribuído, assinale a alternativa correta.

- A) A transparência de acesso oculta diferenças na apresentação de dados e no modo de acesso a um recurso.
- B) A transparência de replicação oculta que um recurso pode ser movido. ✓

para outro computador quando em uso.

C) A transparência de migração oculta que um recurso pode ser compartilhado por diversos usuários, ao mesmo tempo.

D) A transparência de relocação está presente no sistema quando a falha e a recuperação de um recurso são feitas de forma imperceptível ao seu usuário.

E) A transparência de localização permite que um recurso possa ser movido de uma localização para outra.

Resposta:

A transparência de relocação refere-se à capacidade do sistema distribuído de lidar com falhas e recuperação de recursos de forma transparente para o usuário. Isso significa que, se um recurso falhar, o sistema deve ser capaz de recuperá-lo ou realocá-lo sem que o usuário perceba ou seja afetado por essa operação. Essa é uma característica importante para garantir a confiabilidade e disponibilidade do sistema distribuído.

---

(2020, Prefeitura de Vila Velha, analista de infraestrutura) Em Sistemas Distribuídos, o conceito de transparência pode ser aplicado em vários aspectos. O tipo de transparência que esconde do usuário as diferenças da representação de dados e no modo como um recurso é acessado é chamado de:

- A) transparência de localização.
- B) transparência de migração.
- C) transparência de relocação.

D) transparência de acesso.

E) transparência de concorrência.

Resposta:

A transparência de acesso em sistemas distribuídos é aquela que oculta do usuário as diferenças na representação de dados e no modo como

um recurso é acessado. Isso significa que, independentemente de onde os dados estão localizados ou como são acessados (localmente ou remotamente), o usuário percebe apenas uma maneira uniforme de acessar e manipular esses recursos.

---

(2022, FAPESE, analista) O conceito de transparência pode ser aplicado de diferentes aspectos de sistemas distribuídos.

Assinale a alternativa que indica corretamente o conceito de transparência que esconde diferenças na representação dos dados e a forma pela qual recursos podem ser acessados pelo usuário.

- A) Transparência de Acesso.
- B) Transparência de Aplicação.
- C) Transparência de Relocação.
- D) Transparência de Localização.
- E) Transparência de Apresentação.

Resposta:

A transparência de acesso em sistemas distribuídos oculta do usuário as diferenças na representação dos dados e na forma como os recursos podem ser acessados. Isso significa que, independentemente de onde os dados estão armazenados ou como são acessados (localmente ou remotamente), o usuário percebe apenas uma maneira uniforme de acessar e manipular esses recursos.

## ▼ Introdução a processos

### Chamadas de Sistemas

- Se uma aplicação precisa realizar uma instrução privilegiada, ela usa uma chamada de sistema que altera do modo usuário para o modo kernel.
- As chamadas de sistema são a porta de entrada para o modo kernel.
- No caso delas, a interrupção é a nível de software.

### Como as chamadas de sistema são realizadas?

- As chamadas de sistema são realizadas através de instruções Traps.

- Traps são instruções que permitem chamar a atenção do SO, pegar a requisição do usuário e processá-la.
- Traps são conhecidos como interrupções de software.
- Após o término da chamada, a execução continua após a chamada de sistema.

## Interrupção vs Traps

Interrupção:

- Evento externo ao processador.
- Gerada por dispositivos que precisam da atenção do SO.
- Pode não estar relacionada ao processo que está rodando.

Traps:

- Evento inesperado vindo de dentro do processador.
- Causados pelo processo corrente no processador (seja chamada ao So, seja pro instrução ilegal).

## O que é um processo?

Um programa em execução.

## Processo vs programa

Se você abrir dois chromes, você tem dois processos. Mas ainda tem apenas um programa.

Além disso, os processos tem dados de entrada, saída e um estado(executando, bloqueado, pronto).

## Processo em primeiro plano

Interage diretamente com o usuário.

## Processo em segundo plano

Não interage diretamente com o usuário.  
independem dos usuários

## Cada processo possui

- Conjunto de instruções.
- Espaço de endereçamento (Espaço reservado para que o processo possa ler e escrever - 0 até max).
- Contexto de hardware (Valores nos registradores, como PC, ponteiro de pilha, e reg. de prop. gerais).
- Contexto de software (Atributos em geral, como lista de arquivos abertos, variáveis, etc.).

## Espaço de endereçamento

- Texto: código executável dos programas.
- Dados: as variáveis.
- Pilha de execução:
  - Controla a execução do processo.
  - Empilhando chamadas a procedimentos, seus parâmetros e variáveis locais, etc.



## Tabela de processos

- Também chamada de BCP (Bloco de controle de processo).
- Contém informações de contexto de cada processo (ex. ponteiros de arquivos abertos, posição do próximo byte a ser lido em cada arquivo, etc.).
- Contém informações necessárias para trazer o processo de volta, caso o SO tenha que tirá-lo de execução.
- Contém estados de um processo em um determinado tempo.

## Características de processo

- Processos CPU-bound (Orientados a CPU):
  - Processos que utilizam muito processador.
  - O tempo de execução é definido pelos ciclos de processador.
- Processos I/O-bound (Orientados a E/S):

- Tempo de execução é definido pela duração das operações de E/S.
- O ideal é existir um balanceamento entre I/O-bound e CPU-bound.

## Formas de criação de processos

- Inicialização do sistema.
- Execução de uma chamada de sistema para criação de processo, realizada por algum processo em execução.
- Requisição de usuário para criar um novo processo.
- Inicialização de um processo em batch.

## Processos criando outros processos

- No Unix com a função fork(): Cria clone do processo pai (Cópias exatas na memória, mas com identificadores diferentes).
- No Windows com CreateProcess: Cria processo filho, já carregando novo programa nele.

*Trap/Syscall*

## Finalizando processos

- Término normal (Voluntário):
  - A tarefa a ser executada é fimalizada.
  - Ao terminar, o processo executa uma chamada (Comunicando ao SO que terminou): exit (Unix) e ExitProcess (Windows).
- Término por erro (Voluntário):
  - O processo sendo executado não pode ser finalizado. Ex: gcc filename.c; o arquivo filename.c não existe.
- Término por erro fatal (Involuntário):
  - Erro causado por algum err no programa (bug). Ex: Divisão por zero, Referenciaa memoria inexistente...
- Término causado por algum outro processo via chamada a (Involuntário):
  - kill (Unix).
  - TerminateProcess (Windows).

## Estados de um processo

- Executando: Usando a CPU no momento.
- Bloqueado: Incapaz de executar até que um evento externo ocorra.
- Pronto: Em memória, pronto para executar. Apenas esperando uma vaga no processador.

## ▼ Threads

### O que é uma thread?

É uma linha de processamento. Um componente de um processo.

### Características

- Quando você coloca várias threads em um processo, elas dividem o contexto deste processo. Como a memória alocada ao processo.
- Não são protegidos uns contra os outros automaticamente.
- O SO é quem escalona. Então ele é quem faz a ordem.

### Composição

- Identificador de thread
- Registrador PC
- Demais registradores
- Pilha de memória

### Compartilha com threads irmãos

- Seção de código e dados.
- Recursos de SO (ex.: arquivos abertos, sinais)

### Threads são mais leves por quê?

- Oferecem vantagens similares a processos.
- Não requerem típicas técnicas de comunicação de processos.

## Por que usar threads?

- Aumento na capacidade de resposta.
- Economia na alocação de memória e recursos, é mais econômico que a troca de contexto.
- Compartilhamento de recursos.
- Escalabilidade.

## ▼ Exercícios

Q1(ENADE) - Uma alternativa para o aumento de desempenho de sistemas computacionais é o uso de processadores com múltiplos núcleos, chamados multicores. Nesses sistemas, cada núcleo, normalmente, tem as funcionalidades completas de um processador, já sendo comuns, atualmente, configurações com 2, 4 ou mais núcleos. Com relação ao uso de processadores multicores, e sabendo que threads são estruturas de execução associadas a um processo, que compartilham suas áreas de código e dados, mas mantêm contextos independentes, analise as seguintes asserções.

I) Ao dividirem suas atividades em múltiplas threads que podem ser executadas paralelamente, aplicações podem se beneficiar mais efetivamente dos diversos núcleos dos processadores multicores porque:

II) o sistema operacional nos processadores multicores pode alocar os núcleos existentes para executar simultaneamente diversas seqüências de código, sobrepondo suas execuções e, normalmente, reduzindo o tempo de resposta das aplicações às quais estão associadas.

Acerca dessas asserções, assinale a opção correta.

A ) As duas asserções são proposições verdadeiras, e a segunda é uma justificativa correta da primeira.

B) As duas asserções são proposições verdadeiras, mas a segunda não é uma justificativa correta da primeira.

C) A primeira asserção é uma proposição verdadeira, e a segunda, uma proposição falsa.

D) A primeira asserção é uma proposição falsa, e a segunda, uma

proposição verdadeira.

E) Tanto a primeira quanto a segunda asserções são proposições falsas. 

Resposta:

A afirmação I é verdadeira, pois ao dividir as atividades em múltiplas threads que podem ser executadas em paralelo, as aplicações podem aproveitar mais eficientemente os diversos núcleos dos processadores multicores. Isso permite que várias partes do código sejam executadas simultaneamente, aumentando a utilização dos recursos disponíveis.

A afirmação II também é verdadeira. Nos processadores multicores, o sistema operacional pode alocar os núcleos existentes para executar diversas sequências de código de forma simultânea, sobrepondo suas execuções. Isso geralmente resulta em uma redução do tempo de resposta das aplicações, já que várias tarefas podem ser realizadas ao mesmo tempo.

---

Q2 (ENADE) - Uma antiga empresa de desenvolvimento de software resolveu atualizar toda sua

infraestrutura computacional adquirindo um sistema operacional multitarefa, processadores multi-core (múltiplos núcleos) e o uso de uma linguagem de programação com suporte a threads.

O sistema operacional multitarefa de um computador é capaz de executar vários processos (programas) em paralelo. Considerando esses processos implementados com mais de uma thread (multi-threads), analise as afirmações abaixo.

- I. Os ciclos de vida de processos e threads são idênticos. 
- II. Threads de diferentes processos compartilham memória. 
- III. Somente processadores multi-core são capazes de executar programas multi-threads. 
- IV. Em sistemas operacionais multitarefa, threads podem migrar de um processo para outro. 

É correto apenas o que se afirma em

- A) I.
- B) II.
- C) I e III.
- D) I e IV.

**E) II e IV.**

Resposta:

Vamos analisar cada afirmação:

I. Os ciclos de vida de processos e threads são idênticos.

- Esta afirmação é falsa. Embora processos e threads compartilhem algumas características, como os estados (execução, espera, etc.), seus ciclos de vida não são idênticos. Threads podem ser criadas e encerradas dentro de um processo sem necessariamente afetar o ciclo de vida do próprio processo.

II. Threads de diferentes processos compartilham memória.

- Esta afirmação é verdadeira. Threads que pertencem a processos diferentes podem compartilhar memória, o que permite uma comunicação eficiente entre diferentes partes de um sistema.

III. Somente processadores multi-core são capazes de executar programas multi-threads.

- Esta afirmação é falsa. Embora os processadores multi-core sejam adequados para lidar com programas multi-threads devido à capacidade de executar múltiplas threads simultaneamente, processadores de núcleo único também podem executar programas multi-threads, embora não de forma tão eficiente.

IV. Em sistemas operacionais multitarefa, threads podem migrar de um processo para outro.

- Esta afirmação é verdadeira. Em sistemas operacionais multitarefa, threads podem migrar de um processo para outro, geralmente por meio de mecanismos de sincronização e compartilhamento de recursos.

## ▼ Speedup

### O que é o Speedup?

Uma métrica para comparar algoritmos. No caso, sua fórmula é a razão entre o tempo de execução de um algoritmo anterior e o tempo de execução de um algoritmo paralelo.

$$S(n, p) = T(n)/T(n, p)$$

## Concorrência vs Paralelismo

- Paralelismo é quando duas ou mais tarefas são executadas literalmente ao mesmo tempo. Um exemplo é quando tarefas são executadas em núcleos distintos.
- Concorrência ocorre quando duas ou mais tarefas têm o tempo sobreposto, mas não significando que estejam executando necessariamente no mesmo instante. Um exemplo é o pipeline e o escalonador.
- Na prática, não temos controle sobre qual vai ser o caso da execução de um processo.

## ▼ Comunicação entre processos em sistemas distribuídos: forma básica

### Tipos de comunicação em um sistema distribuído

- Comunicação persistente vs transiente.
  - Transiente: O processo nos dois lados (Remetente e destinatário) devem ocorrer ao mesmo tempo.
  - Persistente: O processo nos dois lados (Remetente e destinatário) não precisam ocorrer ao mesmo tempo.
- Comunicação síncrona vs assíncrona.
  - Síncrona: Os processos no remetente e destinatário são sincronizados através de pontos de sincronização. O processo fica bloqueado até obter uma resposta.
  - Assíncrona: Nesse caso, o processo remetente não fica bloqueado após enviar a mensagem. O destinatário ainda pode ficar bloqueado até a mensagem chegar, ou não. *mensagens WhatsApp*

## ▼ Comunicação entre processos em sistemas distribuídos: RPC

## Chamada de processamento remoto (RPC)

- Modelo mais antigo e amigável para programador, consistindo na extensão de chamada de procedimentos convencionais.
- Permite aos programas clientes chamarem procedimentos de forma transparente em servidores.

## Invocação a método remoto (RMI)

- Modelo de programação baseado em objetos, permitindo que estes se comuniquem.
- É uma extensão da invocação a método local que permite a um objeto que está em um processo invocar os métodos de objeto vinculado a outro processo.

## ▼ Resumo para a prova 1

1- Um sistema distribuído é aquele no qual os componentes de hardware ou software, localizados em computadores interligados em rede, comunicam-se e coordenam suas ações apenas enviando mensagens entre si.com base na definição apresentada e nos conceitos básicos de sistemas distribuídos, assinale a opção correta:

- a) A existência de um relógio físico local sincronizado com um relógio global é o que permite aos usuários de um sistema distribuído trocarem mensagens de forma coordenada.
- b) A falha de um componente em um sistema distribuído peer-to-peer causa a interrupção de todos os demais componentes até o seu retorno.
- c) O compartilhamento dos recursos distribuídos entre computadores interligados por uma rede é possível desde que os dispositivos sejam homogêneos em termos de hardware e software.
- d) A adição de novos dispositivos em um sistema distribuído para atender a uma demanda temporária ou crescente está ligada à característica de escalabilidade do sistema.
- e) A possibilidade de vários clientes acessarem de forma concorrente um mesmo recurso compartilhado em um servidor é resultado do alto nível de transparência do sistema.

Resposta:

a) A existência de um relógio físico local sincronizado com um relógio global é o que permite aos usuários de um sistema distribuído trocarem mensagens de forma coordenada.

- Esta afirmação é parcialmente verdadeira. A sincronização de relógios é importante em sistemas distribuídos para coordenar eventos e mensagens, mas é apenas um aspecto da coordenação. Outros mecanismos, como protocolos de comunicação e algoritmos de sincronização, também são necessários para coordenar ações entre os componentes distribuídos.

b) A falha de um componente em um sistema distribuído peer-to-peer causa a interrupção de todos os demais componentes até o seu retorno.

- Esta afirmação é falsa. Em um sistema distribuído peer-to-peer, a falha de um componente geralmente não causa a interrupção de todos os outros componentes. Esses sistemas são projetados para serem resilientes a falhas, e os componentes restantes podem continuar operando de forma independente.

c) O compartilhamento dos recursos distribuídos entre computadores interligados por uma rede é possível desde que os dispositivos sejam homogêneos em termos de hardware e software.

- Esta afirmação é falsa. Em sistemas distribuídos, é possível compartilhar recursos entre computadores interligados, independentemente de serem homogêneos ou heterogêneos em termos de hardware e software. Mecanismos de comunicação e protocolos são desenvolvidos para garantir que diferentes dispositivos possam interagir e compartilhar recursos.

d) A adição de novos dispositivos em um sistema distribuído para atender a uma demanda temporária ou crescente está ligada à característica de escalabilidade do sistema.

- Esta afirmação é verdadeira. A capacidade de adicionar novos dispositivos e aumentar a capacidade de processamento em um sistema distribuído para lidar com demandas crescentes é uma característica importante da escalabilidade do sistema distribuído.

e) A possibilidade de vários clientes acessarem de forma concorrente um mesmo recurso compartilhado em um servidor é resultado do alto nível de transparência do sistema.

- Esta afirmação é falsa. A possibilidade de vários clientes acessarem concorrentemente um mesmo recurso compartilhado em um servidor está mais relacionada à capacidade do sistema distribuído de gerenciar concorrência e sincronização, e não necessariamente à transparência do sistema.

---

2- Alguns dos principais desafios gerais de Sistemas Distribuídos são: escalabilidade, heterogeneidade, segurança, tratamento de falhas, concorrência e escalabilidade. Desses fatores, selecione 3 deles. Explique porque são considerados desafios e apresente estratégias que solucionem ou minimizem problemas relacionados a eles.

Resposta:

### 1. Escalabilidade:

- Desafio: À medida que um sistema distribuído cresce em tamanho ou recebe mais demanda, pode se tornar difícil manter o desempenho. A escalabilidade refere-se à capacidade do sistema de lidar com o aumento da carga de trabalho ou do número de usuários de forma eficiente.
- Estratégias:
  - Distribuição de carga: Distribuir a carga de trabalho entre diferentes nós do sistema para evitar sobrecarga em um único ponto.
  - Arquitetura em camadas: Dividir o sistema em camadas, onde cada camada pode ser escalonada horizontal ou verticalmente conforme necessário.
  - Elasticidade: Capacidade do sistema de adicionar ou remover recursos automaticamente conforme a demanda varia.

### 2. Heterogeneidade:

- Desafio: Os sistemas distribuídos frequentemente envolvem diferentes tipos de hardware, sistemas operacionais e linguagens de programação. Integrar essas diferentes tecnologias pode ser complexo.
- Estratégias:

- Uso de padrões de comunicação e protocolos comuns:  
Padronizar a forma como os diferentes componentes do sistema se comunicam para facilitar a interoperabilidade.
- Uso de middleware: Middleware pode fornecer uma camada de abstração que oculta a heterogeneidade subjacente, permitindo que os diferentes componentes interajam de forma transparente.

### 3. Segurança:

- Desafio: Sistemas distribuídos estão sujeitos a uma série de ameaças de segurança, como acesso não autorizado, interceptação de dados e ataques de negação de serviço.
- Estratégias:
  - Criptografia: Proteger dados confidenciais por meio de técnicas de criptografia para garantir que somente partes autorizadas tenham acesso.
  - Controle de acesso: Implementar políticas de controle de acesso rigorosas para garantir que apenas usuários autorizados tenham permissão para acessar recursos do sistema.
  - Mecanismos de autenticação: Verificar a identidade dos usuários e dos componentes do sistema antes de conceder acesso aos recursos.

### 4. Tratamento de falhas:

- Desafio: Falhas de hardware, software ou de rede são inevitáveis em sistemas distribuídos e podem ter impacto significativo no funcionamento do sistema.
- Estratégias:
  - Tolerância a falhas: Projetar o sistema para continuar funcionando mesmo na presença de falhas. Isso pode envolver o uso de replicação de dados, detecção de falhas e mecanismos de recuperação.
  - Monitoramento: Implementar sistemas de monitoramento que detectem falhas rapidamente e acionem mecanismos de recuperação automaticamente.

### 5. Concorrência:

- Desafio: Em sistemas distribuídos, várias entidades podem tentar acessar e modificar recursos compartilhados simultaneamente, o que pode levar a condições de corrida e inconsistências nos dados.
- Estratégias:
  - Controle de concorrência: Usar técnicas como locks, semáforos e transações para garantir que apenas uma entidade tenha acesso exclusivo a um recurso compartilhado por vez.
  - Coordenação: Implementar protocolos de coordenação para garantir que as operações sejam executadas em uma ordem consistente e que os resultados sejam corretos, mesmo em um ambiente distribuído.

Cada um desses desafios em sistemas distribuídos requer uma abordagem cuidadosa e uma combinação de técnicas e estratégias para serem resolvidos ou mitigados com sucesso.

3- O que é transparência em sistemas distribuídos. Cite todos e explique e exemplifique cada um deles.

Resposta:

Transparência em sistemas distribuídos refere-se à capacidade de ocultar a complexidade do sistema distribuído dos usuários e dos aplicativos que o utilizam. Existem vários tipos de transparência, cada um lidando com diferentes aspectos do sistema distribuído. Abaixo estão os principais tipos de transparência e uma explicação de cada um com exemplos:

### 1. Transparência de Acesso:

- Refere-se à capacidade de acessar recursos distribuídos de forma transparente, como se estivessem localmente disponíveis.
- Exemplo: Ao acessar um arquivo remoto em um sistema distribuído de armazenamento de arquivos, o usuário não precisa se preocupar com a localização física do arquivo ou os detalhes da comunicação de rede. Ele simplesmente acessa o arquivo usando a mesma interface que usaria para acessar um arquivo local.

### 2. Transparência de Localização:

- Oculta a localização física dos recursos distribuídos dos usuários e dos aplicativos.

Drive

- Exemplo: Ao acessar um serviço de banco de dados distribuído, os usuários não precisam saber onde o banco de dados está localizado fisicamente. Eles simplesmente acessam o banco de dados usando um endereço lógico, sem se preocupar com a localização real dos servidores que hospedam o banco de dados.

### 3. Transparência de Migração:

- Permite que recursos distribuídos sejam movidos ou replicados de forma transparente entre diferentes nós do sistema distribuído.
- Exemplo: Um processo em um sistema distribuído pode ser migrado de um nó para outro sem que os usuários ou os aplicativos percebam. Isso pode ser útil para balanceamento de carga ou para lidar com falhas em um nó específico.

### 4. Transparência de Replicação:

- Oculta o fato de que um recurso distribuído está sendo replicado em vários nós do sistema distribuído.
- Exemplo: Um servidor web distribuído pode replicar seu conteúdo em vários servidores para lidar com um grande volume de tráfego. Os usuários não precisam saber que estão acessando uma versão replicada do site; eles simplesmente acessam o site como de costume.

*E-mail, Sisu, Gov*

### 5. Transparência de Concorrência:

- Garante que as operações concorrentes em recursos distribuídos sejam tratadas de forma transparente, sem que os usuários percebam ou tenham que lidar com problemas de concorrência.
- Exemplo: Em um sistema de arquivos distribuído, múltiplos usuários podem acessar e modificar o mesmo arquivo simultaneamente. A transparência de concorrência garante que as operações de leitura e escrita sejam coordenadas de forma transparente para evitar condições de corrida e garantir a consistência dos dados.

*doc - Compartilhado*

### 6. Transparência de Falhas:

- Esconde os efeitos de falhas de hardware ou software dos usuários e dos aplicativos.
- Exemplo: Se um nó em um sistema distribuído falhar, os usuários continuam a acessar os recursos do sistema sem interrupções

percebidas. O sistema distribuído deve ser capaz de detectar e lidar com falhas automaticamente, garantindo que o serviço seja restaurado o mais rápido possível.

Esses são os principais tipos de transparência em sistemas distribuídos, cada um contribuindo para uma experiência de usuário mais simplificada e consistente, independentemente da complexidade subjacente do sistema distribuído.

4- Explique a relação entre processo e thread. Quem monitora esses algoritmos em execução? Dê um exemplo de desafio de comunicação (a) entre processos e (b) entre threads.

Resposta:

Processos e threads são conceitos fundamentais em sistemas operacionais e sistemas distribuídos, ambos relacionados à execução de tarefas e à concorrência de processamento. Aqui está a explicação da relação entre eles e quem os monitora, seguida de exemplos de desafios de comunicação entre processos e threads:

#### 1. Relação entre processo e thread:

- Um processo é uma instância em execução de um programa em um sistema operacional. Cada processo tem seu próprio espaço de endereçamento de memória, recursos de sistema e pelo menos uma thread de execução.
- Uma thread é a menor unidade de processamento que pode ser agendada pelo sistema operacional. Threads compartilham o mesmo espaço de endereçamento de memória dentro de um processo e podem acessar os mesmos recursos do processo.
- Assim, um processo pode ter uma ou várias threads, sendo que as threads dentro do mesmo processo compartilham recursos, como memória e arquivos abertos.

#### 2. Quem monitora processos e threads em execução:

- Em sistemas operacionais, é o kernel que monitora e gerencia processos e threads em execução. O kernel é responsável por atribuir tempo de CPU, lidar com concorrência, gerenciar memória e lidar com operações de entrada/saída.

- O kernel também é responsável por garantir a execução justa e eficiente de processos e threads, além de lidar com eventos como criação, término, escalonamento e sincronização de processos e threads.

### 3. Exemplos de desafios de comunicação:

#### a) Desafio de comunicação entre processos:

- Compartilhamento de dados: Comunicar-se entre processos requer o uso de mecanismos de comunicação interprocessos (IPC), como pipes, filas de mensagens, sockets, entre outros. O desafio é garantir a sincronização e a consistência dos dados compartilhados entre processos.
- Exemplo: Dois processos que precisam trocar dados entre si podem enfrentar problemas de concorrência se tentarem acessar e modificar o mesmo recurso compartilhado ao mesmo tempo, levando a condições de corrida.

#### b) Desafio de comunicação entre threads:

- Sincronização: Threads em um mesmo processo compartilham o mesmo espaço de memória, portanto, compartilham variáveis e recursos. O desafio é garantir que as operações sejam executadas em uma ordem correta e coordenada para evitar condições de corrida e inconsistências nos dados.
- Exemplo: Duas threads que tentam acessar e modificar a mesma variável compartilhada podem enfrentar problemas de concorrência se não forem sincronizadas adequadamente. Isso pode levar a resultados inesperados ou incorretos, como leitura ou gravação de valores inconsistentes.

5- Você trabalha em uma empresa que diariamente precisa processar arquivos de tamanhos diversos, sendo uma importante tarefa do dia a dia da empresa. O algoritmo que foi criado e há anos é usado na empresa é um algoritmo sequencial. Atualmente, alguns arquivos estão na ordem de petabytes de tamanho, o que tem feito com que este processamento leve algumas horas no servidor. Maria, sua colega de equipe, observou que uma parcela do algoritmo sequencial é potencialmente paralelizável. Daí de forma proativa ela fez uma proposta de refatoração e paralelização do algoritmo.

A tabela no quadro apresenta os resultados de speedup feitos pela Maria para uma amostra de arquivos de diferentes tamanhos.

SpeedUp	Tamanho em MBytes	SpeedUp	Tamanho em MBytes
1,047542439	2,6	1,046626842	54,2
1,062245945	3,6	1,056945962	81,7
1,057159687	4,5	1,022983423	86,9
1,050309562	28,5	1,151342626	88,6
1,059230935	36,7	1,017087612	89,8
1,05325509	37,4	1,381273558	145,2
1,049358579	45,9		

Você, como chefe da equipe, visualizou apenas a tabela dos resultados que a Maria experimentou. Diante apenas desses dados, você consegue concluir alguma coisa? Por quê? Que outras informações do experimento você gostaria de obter para auxiliá-lo em uma análise mais completa? Se você estivesse no lugar de Maria, quais os passos metodológicos usados no experimento você forneceria ao seu superior?

mas, pois falta informações na tabela de resultados.

- O algoritmo usado
- A lógica do programa
- O volume de dados usados
- Os tempos de execuções dos programas individualmente
-

Guilherme Xarém Costa Rodrigues

1.

2- Escalabilidade é um desafio, por termos possivel um sistema com muitas rotinas que não eram originalmente desenhadas para elas. Exemplos de rotinas钝iveis, que possuem especificações de muitas rotinas, precisam adicionar serviços extras para essas rotinas, mas sem um exemplo.

Interoperabilidade é quando se resolvem precisar conectar a diversos dispositivos fisicamente da mesma forma. Geralmente é criado uma forma de conectar entre rotinas em cada dispositivo da forma que funciona automaticamente sem fatores dependentes dos dispositivos. Um exemplo seria o WhatsApp em o Google Drive, ambas possuem versões mobile, Desktop que funcionam da forma bem similar como os outros.

Velocidade de fallback é importante por usuários comuns não se importam com precisão sobre porque o sistema falhou, apenas uma mensagem que não está funcionando é melhor do que ele. Isto é como geralmente quando não temos internet ou não conseguimos conectar com algum site, recebemos uma mensagem de que só pode para retornar em outro momento.

a solução?

Concorrência: Quando temos recursos limitados, é necessário forçar os usuários concorrerem por eles, mesmo eles não querendo isso. Isto é como por exemplo no Google Colab, onde só disponibilizam uma CPU ou GPU, que podem ser usadas por um usuário ou vários, mesmo que possam querer que o uso é exclusivo.

2. Em Resumo

3- Transparência em SD é o fato de facilitar a distribuição para o usuário. Temos os seguintes Transparências: acesso, localizações, migrações, relocalizações, salvo, concorrentes, tempo de uso entre outras mais.

Relocalizações: Ocorre quando o usuário desloca máscas que o usuário não precisa saber qual torno está dentro o seu campo de visão se preocupa em se deslocar, pois os turnos são mais precisos e previsíveis em se deslocar, mas as turnos que o usuário não ser tratados garantindo um melhor acesso ao usuário.

Salvo: Quando quando o sistema apresenta um erro, é só redirecionar uma mensagem com um código, quando quando o sistema ativar um salvamento por exemplo.

Migrações: O usuário não precisa saber o princípio físico do seu roteiro para usar, ele usa quando o sistema o google e não sabe nem em qual servidor estou o seu roteiro.

4-a) O processo é uma "instância" de um código, ou seja é a execução de como tempo, ele possui memória própria e protegida pelo sistema Operacional, o próprio sistema operacional monitora e gerencia as rotinas e as comunicações entre elas é normalmente feita por trocas de mensagens.

b) As threads são subdivisões de um processo, cada processo possui diferentes threads, elas possuem a memória compartilhada e são gerenciadas pelo observador, mas é o próprio sistema operacional que monitora elas durante a execução disponibilizando a não exclusividade, mas elas comunicam entre si para elas usarem a mesma parte da memória, pode haver conflito entre elas em alguns casos e devem ser tratados pelo próprio observador.

Guilherme Ximenes Costa Rodrigues

5. Não posso considerar modo, pois os faltas de informações  
do tempo faz que no algoritmo o es texto, podem estar  
relacionado dados com algum tipo de regras.

Sugestão de um resumo do como foi feito a probabilização:  
Sugestão de um resumo do como foi feito a probabilização:  
possível de saber quantos dias cada código foi usado  
ou houve ou não remoção de outliers, as especificações  
da máquina e se possuem informações do uso da  
máquina e se possuem informações do uso das  
máquinas durante a execução para ver se os recursos  
estão realmente sendo bem utilizados.

-- Eu:

1º Faria uma tabela com pelo menos 50 dados por tipo  
de arquivos, armazenando os valores de tempo.

2º mostraria a remoção de algum outlier, e realizaria  
as médias com os valores restantes.

3º Faria os cálculos de SpeedUp (Versão Anterior)  
+ armazena os valores médios (Versão Nova)

4º Faria um resumo mostrando como foi feita a proba-  
bilização e minhas expectativas

5º Faria um resumo mostrando os cálculos e se  
conseguiu ou não melhorar a performance. C

BOA PROVA!

- 1. (0,3 ponto)** (Questão ENADE 2021) Um sistema distribuído é aquele no qual os componentes de hardware ou software, localizados em computadores interligados em rede, comunicam-se e coordenam suas ações apenas enviando mensagens entre si.

COULOURIS, G. et al. *Sistemas distribuídos: conceitos e projeto*. 5. ed., Porto Alegre: Bookman, 2013 (adaptado).

Com base na definição apresentada e nos conceitos básicos de sistemas distribuídos, assinale a opção correta.

- a) A existência de um relógio físico local sincronizado com um relógio global é o que permite aos usuários de um sistema distribuído trocarem mensagens de forma coordenada.

- b) A falha de um componente em um sistema distribuído peer-to-peer causa a interrupção de todos os demais componentes até o seu retorno.

- c) O compartilhamento dos recursos distribuídos entre computadores interligados por uma rede é possível desde que os dispositivos sejam homogêneos em termos de hardware e software.

- d) A adição de novos dispositivos em um sistema distribuído para atender a uma demanda temporária ou crescente está ligada à característica de escalabilidade do sistema.

- e) A possibilidade de vários clientes acessarem de forma concorrente um mesmo recurso compartilhado em um servidor é resultado do alto nível de transparência do sistema.

- 2. (0,5 ponto)** Alguns dos principais desafios gerais de Sistemas Distribuídos são: escalabilidade,

heterogeneidade, segurança, tratamento de falhas, concorrência e escalabilidade. Desses fatores, selecione 3 deles. Explique porque são considerados desafios e apresente estratégias que solucionem ou minimizem problemas relacionados a eles. *Vou considerar a questão extra.*

- 3. (0,5 ponto)** O que é transparência em sistemas distribuídos. Cite 3 exemplos e explique e exemplifique cada um deles.

- 4. (0,5 ponto)** Explique a relação entre processo e thread. Quem monitora esses algoritmos em execução? Dê um exemplo de desafio de comunicação (a) entre processos e (b) entre threads.

- 5. (0,7 ponto)** Você trabalha em uma empresa que diariamente precisa processar arquivos de tamanhos diversos, sendo uma importante tarefa do dia a dia da empresa. O algoritmo que foi criado e há anos é usado na empresa é um algoritmo sequencial. Atualmente, alguns arquivos estão na ordem de petabytes de tamanho, o que tem feito com que este processamento leve algumas horas no servidor. Maria, sua colega de equipe, observou que uma parcela do algoritmo sequencial é potencialmente paralelizável. Daí de forma proativa ela fez uma proposta de refatoração e paralelização do algoritmo.

A tabela no quadro apresenta os resultados de speedup feitos pela Maria para uma amostra de arquivos de diferentes tamanhos.

SpeedUp	Tamanho em MBytes	SpeedUp	Tamanho em MBytes
1,047542439	2,6	1,046626842	54,2
1,062245945	3,6	1,056945962	81,7
1,057159687	4,5	1,022983423	86,9
1,050309562	28,5	1,151342626	88,6
1,059230935	36,7	1,017087612	89,8
1,05325509	37,4	1,381273558	145,2
1,049358579	45,9		

Você, como chefe da equipe, visualizou apenas a tabela dos resultados que a Maria experimentou. Diante apenas desses dados, você consegue concluir alguma coisa? Por quê? Que outras informações do experimento você gostaria de obter para auxiliá-lo em uma análise mais completa?

Se você estivesse no lugar de Maria, quais os passos metodológicos usados no experimento você forneceria ao seu superior?

## Sistemas Paralelos e Distribuídos - RESUMO - PROVA 01

**Programação Sequencial:** Um programa é considerado programação sequencial quando este é visto como uma série de instruções sequenciais que devem ser executadas num único processador.

**Programação Paralela:** Um programa é considerado programação paralela quando este é visto como um conjunto de partes que podem ser resolvidas **concorrentemente**. Cada parte é igualmente constituída por uma série de instruções sequenciais, mas que no seu conjunto podem ser executadas simultaneamente em vários processadores.

principais motivos para utilizar programação paralela são:

- Reduzir o tempo necessário para solucionar um problema
- Resolver problemas mais complexos e de maior dimensão

paralelização → modelos associados à memória compartilhada

distribuição → modelos associados à troca de mensagens

### Sistemas Distribuídos

O sistema distribuído consiste em computadores independentes e se apresentam como um único sistema (middleware) para os usuários.

middleware é o software que conecta programas separados existentes;  
Normalmente fica entre a aplicação e o sistema operacional.

Uma característica importante é que as diferenças entre os vários computadores e o modo como eles se comunicam estão, em grande parte, ocultas ao usuário.

A principal meta de um sistema distribuído é facilitar aos usuários, e às aplicações, o acesso aos recursos remotos e o seu compartilhamento de maneira controlada e eficiente. (fácil acesso a recursos, permitir expansão, transparência)

Há vários cenários em que vemos um sistema distribuído:

- Computadores conectados por meio de uma rede e separados por qualquer distância

**Transparência:** Um sistema distribuído que é capaz de se apresentar para o usuário como um único sistema é denominado transparente

Tipos de transparência:

- Transparência de acesso: permite acesso a componentes remotos e locais através das mesmas operações  
(Oculta diferenças na representação de dados e no modo de acesso a um recurso)

Ex.: Google Drive

- Transparência de localização: permite acesso a componentes sem conhecimento da sua localização física – Existe um mecanismo que determina a localização baseada em um nome  
(Oculta o lugar onde o recurso está localizado)  
Ex.: URL
- Transparência de migração: um recurso é movido de um local para outro sem que o usuário precise estar ciente. Normalmente, é movido de maneira planejada e controlada, como migração de dados entre servidores ou migração de máquinas virtuais entre hosts.  
(Oculta que um recurso pode ser movido para outra localização)
- Transparência de realocação: um recurso é movido mesmo em uso e sem que o usuário perceba a mudança, como uma conexão de rede ou uma aplicação em execução.  
(Oculta que um recurso pode ser movido para outra localização enquanto ele está sendo usado)
- Transparência de replicação: o usuário desconhece a existência de várias cópias do recurso  
(Oculta que um recurso é replicado)
- Transparência de falha: a presença de falhas não é percebida pelos usuários  
(Oculta a falha e a recuperação de um recurso)
- Transparência de concorrência: permitir que recursos compartilhados sejam acessados de forma segura e consistente, evitando condições de corrida, inconsistências nos dados e outros problemas relacionados à concorrência.  
(Oculta que um recurso pode ser compartilhado por diversos usuários)

## Exercícios

Entre os vários tipos de transparência que os sistemas distribuídos podem fornecer, o ocultamento do fato de que há várias cópias de um recurso disponíveis no sistema é conhecido como:

- (A) transparência de acesso.
- (B) transparência de transação.
- (C) transparência de replicação.
- (D) transparência de concorrência
- (E) transparência de migração.

O requisito no qual os programas clientes não devem conhecer a distribuição de arquivos, sendo que um único conjunto de operações é fornecido para acesso a arquivos locais e remotos, é denominado transparência de

- A) acesso.
- B) desempenho.
- C) escala.
- D) localização.
- E) mobilidade.

Sistemas distribuídos são aqueles dispostos em computadores distintos e autônomos, que trabalham juntos para dar a impressão de ser um único sistema, com acesso local a seus usuários. Uma característica importante nesses sistemas é a transparência. Acerca das formas de transparência de um sistema distribuído, assinale a alternativa correta.

- A) A transparência de acesso oculta diferenças na apresentação de dados e no modo de acesso a um recurso.
- B) A transparência de replicação oculta que um recurso pode ser movido para outro computador quando em uso.
- C) A transparência de migração oculta que um recurso pode ser compartilhado por diversos usuários, ao mesmo tempo.
- D) A transparência de relocação está presente no sistema quando a falha e a recuperação de um recurso são feitas de forma imperceptível ao seu usuário.
- E) A transparência de localização permite que um recurso possa ser movido de uma localização para outra.

Em Sistemas Distribuídos, o conceito de transparência pode ser aplicado em vários aspectos. O tipo de transparência que esconde do usuário as diferenças da representação de dados e no modo como um recurso é acessado é chamado de:

- A) transparência de localização.
- B) transparência de migração.
- C) transparência de relocação.
- D) transparência de acesso.
- E) transparência de concorrência.

O conceito de transparência pode ser aplicado de diferentes aspectos de sistemas distribuídos. Assinale a alternativa que indica corretamente o conceito de transparência que esconde diferenças na representação dos dados e a forma pela qual recursos podem ser acessados pelo usuário.

- A) Transparência de Acesso.
- B) Transparência de Aplicação.
- C) Transparência de Relocação.
- D) Transparência de Localização.
- E) Transparência de Apresentação.

**Tipos de sistemas distribuídos:**

**Sistemas de computação de cluster:** Um cluster utiliza vários sistemas de computador que são idênticos em hardware e sistema operacional.

Tornaram-se populares quando a razão preço/desempenho de computadores pessoais e estações de trabalho melhorou.

A computação de cluster é usada para programação paralela, na qual um único programa é executado em paralelo em várias máquinas.

**Sistemas de computação em grade:** Sistemas de computador heterogêneos, podem ser diferentes o hardware, sistemas operacionais, redes, domínios administrativos, políticas de segurança e assim por diante

**Sistemas distribuídos pervasivos:** Pequenos equipamentos de computação como objetos diários que mal são notados; ausência geral de controle administrativo humano; os dispositivos se juntam ao sistema para acessar/fornecer informações; Requer meios para ler, armazenar, gerenciar e compartilhar informação com facilidade

**Sistemas domésticos:** São compostos por um ou mais computadores pessoais; integram eletrônicos de consumo típicos como aparelhos de tv, equipamentos de áudio e vídeo, dispositivos para jogos, smartphones etc esses dispositivos serão conectados a um único sistema distribuído

Explique o que quer dizer transparência (de distribuição) e dê exemplos de diferentes tipos de transparência.

Explique como funcionaria um sistema escalável.

A troca de mensagens é um dos principais fatores que afetam o desempenho de um sistema distribuído:

Pesquise as velocidades de comunicação dos seguintes padrões. Indique e justifique o tipo de SD mais indicado para o seu uso:

- a) Ethernet
- b) Wi-Fi
- c) ZigBee
- d) InfiniBand
- e) Bluetooth
- f) Fibra

Com o máximo de detalhes que souber, esboce um projeto para um sistema multimídia em uma Smart Home. O sistema é composto de um servidor de mídia e um cliente sem fio.

O cliente é quem possui o equipamento (análogo) de áudio e vídeo.

O servidor não possui teclado ou monitor conectados a ele.

### Threads

Thread (linha de execução) é uma sequência de instruções que faz parte de um processo principal. Um software é organizado em processos. Cada processo é dividido em threads, que formam tarefas independentes, mas relacionadas entre si.

Os sistemas operacionais organizam as tarefas a serem executadas em processos, que são divididos em um ou mais threads.

Cada processo é uma sequência de instruções que está ligada a um software. Se um processo está sendo executado por um núcleo do processador, significa que o software ligado a ele está em execução. Os threads formam conjuntos menores de instruções dentro de uma tarefa maior.

## vantagens das threads

Aumento de velocidade, Maior eficiência, Compartilhamento de recursos

O que o método join faz?

Uma alternativa para o aumento de desempenho de sistemas computacionais é o uso de processadores com múltiplos núcleos, chamados multicores. Nesses sistemas, cada núcleo, normalmente, tem as funcionalidades completas de um processador, já sendo comuns, atualmente, configurações com 2, 4 ou mais núcleos. Com relação ao uso de processadores multicores, e sabendo que threads são estruturas de execução associadas a um processo, que compartilham suas áreas de código e dados, mas mantêm contextos independentes, analise as seguintes asserções.

I) Ao dividirem suas atividades em múltiplas threads que podem ser executadas paralelamente, aplicações podem se beneficiar mais efetivamente dos diversos núcleos dos processadores multicores porque

II) o sistema operacional nos processadores multicores pode alocar os núcleos existentes para executar simultaneamente diversas seqüências de código, sobrepondo suas execuções e, normalmente, reduzindo o tempo de resposta das aplicações às quais estão associadas.

Acerca dessas asserções, assinale a opção correta.

- A ) As duas asserções são proposições verdadeiras, e a segunda é uma justificativa correta da primeira.
- B) As duas asserções são proposições verdadeiras, mas a segunda não é uma justificativa correta da primeira.
- C) A primeira asserção é uma proposição verdadeira, e a segunda, uma proposição falsa.
- D) A primeira asserção é uma proposição falsa, e a segunda, uma proposição verdadeira.
- E) Tanto a primeira quanto a segunda asserções são proposições falsas.

Uma antiga empresa de desenvolvimento de software resolveu atualizar toda sua infraestrutura computacional adquirindo um sistema operacional multitarefa, processadores multi-core (múltiplos núcleos) e o uso de uma linguagem de programação com suporte a threads.

O sistema operacional multitarefa de um computador é capaz de executar vários processos (programas) em paralelo. Considerando esses processos implementados com mais de uma thread (multi-threads), analise as afirmações abaixo.

- I. Os ciclos de vida de processos e threads são idênticos.
- II. Threads de diferentes processos compartilham memória.
- III. Somente processadores multi-core são capazes de executar programas multi-threads.
- IV. Em sistemas operacionais multitarefa, threads podem migrar de um processo para outro.

É correto apenas o que se afirma em

- A) I.
- B) II.
- C) I e III.
- D) I e IV.
- E) II e IV.

Paralelismo ocorre quando duas ou mais tarefas são executadas literalmente ao mesmo tempo. Ex: tarefas executadas em núcleos de processamento distintos

Concorrência ocorre quando duas ou mais tarefas têm o tempo sobreposto, mas não significando que estejam executando necessariamente no mesmo instante. Ex: pipeline

“Concorrência é sobre lidar com várias coisas ao mesmo tempo.

Paralelismo é sobre fazer várias coisas ao mesmo tempo.”

Speedup é a razão do tempo de execução do algoritmo anterior  $T(n)$  com o tempo de execução do algoritmo paralelo  $T(n,p)$

$$S(n,p) = T(n)/T(n,p)$$

Diferenças entre threads e processos

- Threads, por padrão, possui compartilhamento de memória, enquanto os processos, por padrão, não possui
- Início/troca de tarefa é computacionalmente menos custosa para as threads e mais custosa para os processos
- As threads requerem poucos recursos, enquanto os processos querem mais
- As threads necessita de mecanismos de sincronização para manipular os dados corretamente, para os processos, a sincronização não é necessária

Comunicação entre processos (sistemas distribuídos)

A comunicação em sistemas distribuídos é sempre baseada em **troca de mensagens** de baixo nível entre processos como a oferecida pela rede subjacente

troca de mensagens de forma básica: UDP, TCP, multicasting

troca de mensagens em mais alto nível (middleware): RPC (comunicação cliente-servidor) RMI (comunicação entre objetos distribuídos)

Quais as características da comunicação entre processos que utilizam protocolos UDP? E quais as relacionadas ao TCP?

## O que é multicast?

Características da comunicação entre processos

A passagem de mensagens entre um par de processos é feita a partir de duas operações de comunicação: send e receive

Esta atividade pode implicar na sincronização dos dois processos

comunicação persistente vs. transitória

transitória: remetente descarta a mensagem se ela não puder ser entregue ao destinatário

persistente: a mensagem é guardada pelo tempo que for necessário até que seja possível entregar ao destinatário

comunicação síncrona vs. assíncrona

síncrona: processos remetente e destino são sincronizados a cada mensagem (quando send e receive são bloqueantes)

pontos de sincronização: remetente bloqueado

- no envio da requisição
- no processamento da requisição
- na entrega da requisição

problemas: o cliente fica parado enquanto estiver esperando a resposta; falhas devem ser tratadas com rapidez; há serviços que não se encaixam neste modelo

assíncrona: processo remetente não fica bloqueado, já o processo destino pode ser bloqueante e não-bloqueante.

A arquitetura cliente/servidor é normalmente baseada no modelo de comunicação transitória síncrona

comunicação via sockets

comunicação entre dois processos que estejam na mesma máquina ou na rede(TCP/IP)

em protocolos Internet as mensagens são enviadas para destinos definidos pelo par endereço IP e porta local

ip:porta -> `127.0.0.1:4477`

comunicação entre processos via middleware

## RPC - chamadas de processamento remotos

Procedimentos são estruturas de dados que operam isoladamente, como caixas pretas

- Dada uma chamada, executa alguma ação!

Por que não executar procedimentos em máquinas separadas?

- O cliente chama o procedimento de forma transparente
- O servidor executa essa chamada de forma transparente
- Algo implementa essa transparência:

middleware de comunicação

Daí surgiu a ideia do RPC

vantagens do RPC:

**Processo**: programa em execução

↳ memória compartilhada

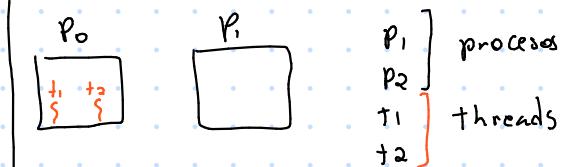
↳ processos conversando entre si (chamadas do Sistema Operacional)

concorre entre si  
pelo processador

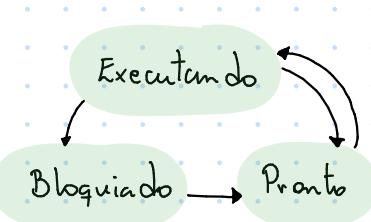
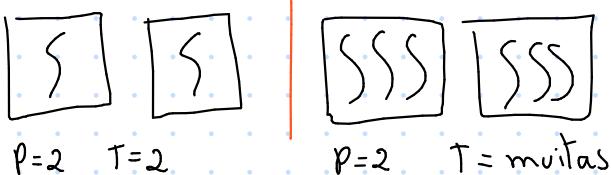
Concurrente x Paralelo

2 ou mais processos mais  
executando em 2 ou mais  
núcleos diferentes (processos/thread)  
o mais independente possível

**Threads**: Processo leve (alocação de recurso e troca de contexto)

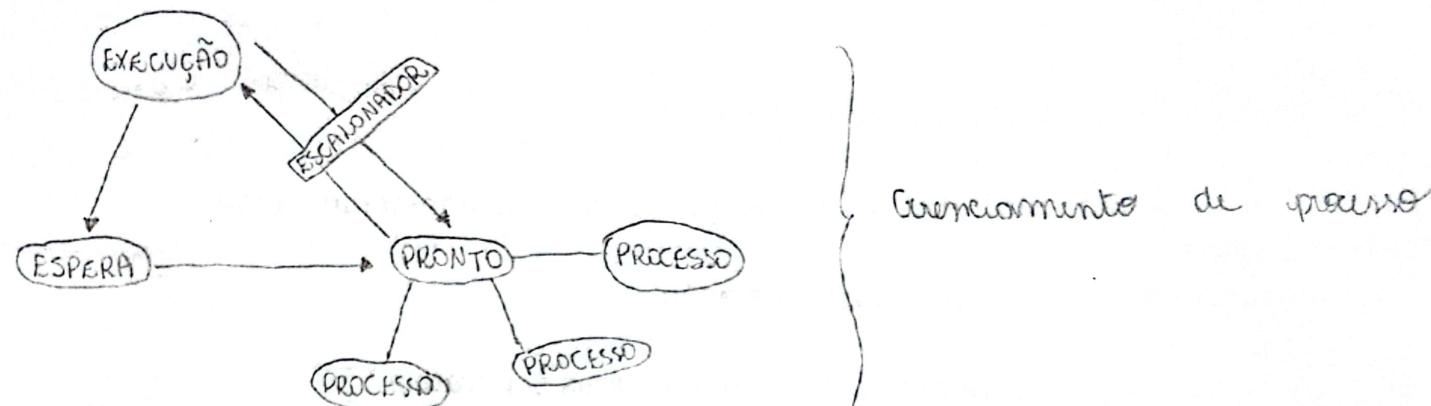


→ thread principal ou thread de controle ou processo



Status de um  
processo

Processo: programa em execução



Iniciação do processo: inicio do SO;  
início de um job;  
execução de chamada de sistema;  
solicitação do usuário para criar um novo processo

Sinais: não manobras pré-definidas; são uma espécie de interrupção ao processo corrente

O processo receptor conhece apenas o tipo de sinal, sem efetivamente conhecer o emitir desse sinal

Se receber esse sinal, o processo pede interromper sua execução e devolver para tratador (handler)

O sinal pode ser ignorado; (caso para sinais SIGKILL e SIGSTOP)

Ou conter no comportamento default do SO

abrir o processo,  
suspenso o processo,  
continuar a execução do processo.

POSIX: é possível bloquear seletivamente a entrega de sinais a um processo

handler)

O sinal pode ser ignorado; (não para sinais SIGKILL e SIGSTOP)

ou confiar no comportamento default do SO

- abrir o processo;
- suspender o processo;
- continuar a execução do processo.

POSIX: é possível bloquear seletivamente a entrega de sinais a um processo

Pipes: constituem um canal de comunicação unidirecional entre processos com um endereço em comum. Uma vez estabelecido o pipe entre os processos, um deles pode enviar mensagens para o outro. O envio e recebimento destas mensagens é feito com o modus `Write`

Recurso utilizado para tratar entrada e saída de dados

Um pipe é criado pela chamada de sistema

Após o pipe é feito a chamada `fork()`, criando um canal de comunicação entre pai e filho

`fork()` → função do SO que cria a cópia exata do processo atual e retorna o ID do processo filho.

Os dois processos são completamente independentes, com PIDs diferentes, mas têm exatamente o mesmo código

... essa é uma forma transparente

- Algo implementa essa transparência:

middleware de comunicação  
Daí surgiu a ideia do RPC

vantagens do RPC: transparência, facilidade de desenvolvimento,

limites do RPC: não suporta interação assíncrona entre cliente e servidor; diferentes implementações costumam ser incompatíveis

RPC: permite que um computador cliente solicite a execução de um procedimento de um computador servidor.

funcionamento: o cliente envia uma mensagem contendo o identificador do procedimento, seus parâmetros e um espaço reservado para o valor retornado

O servidor recebe a mensagem, executa o procedimento solicitado e envia a resposta de volta para o cliente

## **Introdução e objetivos de SD**

- O que são sistemas distribuídos
- Metas: compartilhamento de recursos, segurança, transparência, escalabilidade

## **Tipos de Sistemas Distribuídos**

- Cluster, Grid, Cloud, Pervasivo

## **Transparência em SD**

- Acesso, Localização, Migração, Realocação, Replicação, Concorrência, Falhas

## **Processos e Threads**

- Conceito de processo (BCP, estados, contexto)
- Criação/finalização de processos (fork/exec, CreateProcess)
- Threads: estrutura, criação, join, compartilhamento de memória
- Concorrência vs. Paralelismo

## **Comunicação entre processos**

- Síncrona x Assíncrona
- Persistente x Transiente
- Sinais, pipes, sockets, RPC, RMI

## **Programação Paralela**

- Speedup, overhead, warm-up, outliers
- Medição de desempenho, ponto ótimo de threads

## **1. Conceitos Fundamentais**

- O que são Sistemas Distribuídos? (Definição, características)
- Diferença entre paralelismo e concorrência
- Tipos de sistemas distribuídos: *Cluster, Grid, Pervasivo*
- Diferença entre sistemas paralelos e distribuídos (memória compartilhada x troca de mensagens)

## 2. Transparência em SD

Estude com atenção os **tipos de transparência**:

- Acesso
- Localização
- Migração
- Realocação
- Replicação
- Concorrência
- Falhas

Exercícios relacionados aparecem bastante em concursos e provas!

## 3. Escalabilidade e Metas de um SD

- Conceito de escalabilidade
- Desafios: gargalos, serviços centralizados, latência
- Estratégias: balanceamento de carga, elasticidade

## 4. Processos e Threads

- O que é um processo? E uma thread?
- Diferenças entre processos e threads
- Contexto de hardware/software de um processo
- Criação e finalização de processos (`fork`, `CreateProcess`)
- Vantagens e desafios de usar threads
- Uso de `threading` em Python

## 5. Comunicação entre Processos

- Comunicação síncrona vs assíncrona
- Comunicação persistente vs transitória
- Sinais, fluxos padrão, sockets
- Uso de `subprocess` em Python

## 6. Speedup e Benchmark

- Cálculo de Speedup:  $S(n, p) = T(n) / T(n, p)$
- Importância de descartar **warm-ups** e **outliers**
- Influência do número de threads no desempenho

## 7. RPC – Remote Procedure Call

- Conceito de RPC e RMI

- Componentes: cliente, servidor, *stub*
  - Vantagens para sistemas distribuídos
- 

## Resumo Completo – Sistemas Paralelos e Distribuídos (SPD)

---

### 1. O que são Sistemas Distribuídos?

São coleções de computadores independentes, interconectados, que se apresentam aos usuários como um sistema único e coerente.

#### Objetivos:

- Compartilhamento de recursos
  - Transparência
  - Escalabilidade
  - Segurança
  - Flexibilidade
  - Redução de comunicação indesejada
- 

### 2. Diferença entre Sistemas Paralelos e Distribuídos

Paralelismo	Distribuição
Memória compartilhada	Troca de mensagens entre máquinas
Tarefas executadas ao mesmo tempo	Tarefas executadas em sistemas separados
Usado em multi-core	Usado em redes de computadores

---

### 3. Tipos de Sistemas Distribuídos

#### Cluster Computing

- Ambiente homogêneo (mesmo SO e hardware)

- Forte acoplamento
- Um nó mestre distribui tarefas

### Grid Computing

- Hardware heterogêneo (instituições distintas)
- Cooperação entre organizações
- Usado em pesquisa

### Cloud Computing

- Recurso como serviço (VM, app, armazenamento)
- Escalável sob demanda

### Pervasivo (Ubiquidade)

- Computação embarcada, móvel
- Pequenos dispositivos
- Redes instáveis, uso de bateria

---

## 4. Transparência em Sistemas Distribuídos

Permite esconder detalhes da distribuição do sistema.

Tipo	O que oculta/exemplo
Acesso	Diferenças de representação e acesso a recursos (ex: Google Drive)
Localização	Local físico do recurso (ex: URL)
Migração	Recurso pode ser movido sem o usuário saber (ex: mover VM entre servidores)
Realocação	Recurso pode ser movido <b>enquanto</b> está em uso (ex: vídeo em streaming)
Replicação	Existência de múltiplas cópias de um recurso (ex: CDN)
Concorrência	Acesso simultâneo aos recursos sem conflitos (ex: sistema de arquivos distribuído)
Falha	Oculta falhas e recuperação automática (ex: serviço que continua após queda de nó)



## 5. Escalabilidade

### Definição:

Capacidade de crescer (nós, usuários, carga) sem perda de desempenho.

### Desafios:

- Evitar gargalos (ex: ponto único de falha)
- Controlar custo
- Ocultar latência

### Técnicas:

- Balanceamento de carga
  - Distribuição geográfica
  - Aumento de recursos sob demanda (elasticidade)
- 



## 6. Processos e Threads

### Processo:

Programa em execução com contexto (PC, registradores, memória).

### Thread:

Unidade de execução dentro de um processo.

### Diferenças:

- Threads compartilham memória e recursos do processo pai
- Processos têm memória isolada

### Vantagens de threads:

- Menor custo que processos
- Melhor resposta
- Compartilhamento de recursos
- Escalabilidade com multicores

## Criação:

- Unix: `fork()`
  - Windows: `CreateProcess()`
  - Java/Python: `threading.Thread()`
- 

## 7. Concorrência vs Paralelismo

Concorrência	Paralelismo
Tarefas se sobrepõem no tempo	Tarefas executam literalmente ao mesmo tempo
Simulado por escalonamento	Requer múltiplos núcleos
Ex: sistema multitarefa	Ex: execução de threads em vários núcleos

---

## 8. Comunicação entre Processos (IPC)

### Tipos:

- **Sinais (Signal)**: eventos entre processos
- **Fluxos padrão (stdin/stdout)**: entrada e saída redirecionada
- **Sockets**: comunicação entre processos locais ou remotos
- **Pipes**: canais de comunicação em sistemas Unix

### Classificações:

- **Síncrona**: bloqueia até resposta (ex: chamada remota)
  - **Assíncrona**: envio não bloqueante
  - **Persistente**: mensagens podem ser entregues após tempo
  - **Transiente**: remetente e receptor precisam estar ativos
- 

## 9. RPC (Remote Procedure Call)

Permite chamada de funções em outro computador como se fossem locais.

### Componentes:

- Cliente
- Servidor
- Stubs (intermediários que empacotam e desempacotam dados)

**Facilita:**

- Comunicação entre processos remotos
  - Abstração da rede para o programador
- 



## 10. Speedup

**Fórmula:**

$$\text{Speedup} = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}} = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}$$

**Importância:**

Avalia ganho de desempenho ao usar múltiplas threads/processos.

**Atenção a:**

- **Warm-up:** execuções iniciais não confiáveis
- **Outliers:** valores fora do padrão devem ser removidos
- **Overhead:** tempo extra para gerenciar paralelismo

**Número ótimo de threads:**

Depende de:

- Número de núcleos disponíveis
  - Custo de sincronização
  - Tarefas CPU-bound vs I/O-bound
- 



## 11. Desafios em Sistemas Distribuídos

Desafio	Estratégias
Escalabilidade	Balanceamento de carga, arquitetura em camadas, elasticidade

Heterogeneidade	Uso de middleware e protocolos padrões
Segurança	Criptografia, autenticação, controle de acesso
Concorrência	Locks, semáforos, transações, coordenação
Falhas	Tolerância a falhas, replicação, monitoramento
Tempo de resposta	Minimizar latência, evitar gargalos, otimização de comunicação

---

## 1. O que são Sistemas Distribuídos

### Definição:

Um **Sistema Distribuído (SD)** é um conjunto de computadores autônomos interligados por uma rede que apresenta, para o usuário final, ser um sistema único e coerente.

“É como se vários computadores trabalhassem juntos escondendo do usuário que estão separados.”

---

### Objetivos:

1. **Compartilhamento de recursos** (dados, arquivos, impressoras, processamento)
  2. **Transparência** (usuário não precisa saber onde os recursos estão)
  3. **Escalabilidade** (crescer sem perder desempenho)
  4. **Segurança** (controle de acesso, criptografia)
  5. **Tolerância a falhas** (se um nó falhar, o sistema continua funcionando)
  6. **Flexibilidade** (fácil expansão ou modificação)
- 

### Exemplos:

- **Google Drive**: arquivos armazenados em servidores ao redor do mundo, mas o usuário vê como se fosse uma pasta local.
- **Netflix**: vários servidores ao redor do mundo entregando o vídeo de forma transparente.
- **Sistemas bancários**: processam transações em várias localidades, mas o usuário vê

como uma conta única.

---

### Comunicação em SD:

A comunicação entre os computadores é feita **exclusivamente por troca de mensagens** (nada de memória compartilhada).

---

### Características dos algoritmos descentralizados:

- Nenhuma máquina tem a visão completa do sistema
  - Máquinas tomam decisões locais
  - Não depende de relógio global
  - Falhas em uma máquina não afetam o sistema todo
- 

## 2. Diferença entre Sistemas Paralelos e Distribuídos

---

### Conceito-chave:

	Sistemas Paralelos	Sistemas Distribuídos
<b>Localização física</b>	Múltiplos núcleos/processadores no <b>mesmo computador</b>	Computadores separados (cada um com seu próprio hardware)
<b>Memória</b>	<b>Compartilhada</b> entre os núcleos	<b>Independente</b> ; cada máquina tem sua própria memória
<b>Comunicação</b>	Através de <b>variáveis compartilhadas</b> (threads)	Por <b>mensagens</b> (rede)
<b>Objetivo</b>	Alta performance para <b>um problema específico</b>	Integração de recursos geograficamente distribuídos
<b>Exemplo</b>	Algoritmos com threads em um PC	Sistema bancário, Google, Netflix

---

## Explicando com exemplos:

- **Paralelismo:** Seu programa Java usando `8 threads` no seu PC com processador octa-core = *paralelismo com memória compartilhada*.
  - **Distribuição:** Um servidor no Brasil e outro na Alemanha trocando mensagens via internet para processar requisições = *distribuição via rede*.
- 

## Quando os dois se misturam?

Na prática, um **sistema distribuído** pode ter **nós paralelos** (cada nó com vários núcleos), e um **algoritmo paralelo** pode envolver comunicação entre máquinas (por exemplo, em clusters).

---

## Dica para a prova:

"A paralelização está geralmente associada à memória compartilhada, enquanto a distribuição à troca de mensagens."

(*Frase clássica usada em aulas e questões de prova*)

---

## 3. Tipos de Sistemas Distribuídos

Sistemas distribuídos podem ser categorizados em diferentes modelos, conforme as características do hardware, software e o modo como os recursos são gerenciados. A seguir, veja os principais tipos:

---

### 1. Cluster Computing

- **Descrição:** Um conjunto de computadores interligados, geralmente idênticos ou muito parecidos em termos de hardware e sistema operacional.
- **Características:**
  - **Homogeneidade:** Os nós são semelhantes, facilitando a administração e o balanceamento de carga.

- **Comunicação:** Geralmente ocorre através de uma rede local (LAN), com baixa latência.
  - **Arquitetura:** Um nó mestre pode coordenar as tarefas ou um algoritmo paralelo pode distribuir a carga uniformemente entre os nós.
  - **Exemplo:** Um cluster de servidores em um centro de dados que executa cálculos científicos ou simulações intensivas.
- 

## 2. Grid Computing (Computação em Grade)

- **Descrição:** Integração de recursos computacionais heterogêneos (computadores com diferentes hardware, sistemas operacionais e localizações) para resolver problemas complexos.
  - **Características:**
    - **Heterogeneidade:** Os recursos podem pertencer a organizações diferentes e possuir características variadas.
    - **Interoperabilidade:** Uso de padrões e middleware para unir diferentes plataformas de forma transparente.
    - **Colaboração:** Muitas vezes envolve instituições de pesquisa ou empresas que compartilham recursos para objetivos comuns.
  - **Exemplo:** Projetos científicos que reúnem computadores de universidades e centros de pesquisa ao redor do mundo para simulações de modelos climáticos ou análises genéticas.
- 

## 3. Cloud Computing (Computação em Nuvem)

- **Descrição:** Fornece recursos computacionais como serviços sob demanda, usando infraestrutura virtualizada.
- **Características:**

- **Escalabilidade:** Capacidade de adicionar ou remover recursos conforme a demanda do usuário.
  - **Elasticidade:** Cobrança baseada no consumo, permitindo otimização de custos.
  - **Abstração:** O usuário não necessita conhecer a localização física dos dados ou a infraestrutura por trás do serviço.
  - **Exemplo:** Serviços como Amazon Web Services (AWS), Google Cloud Platform e Microsoft Azure, que oferecem servidores virtuais, armazenamento e aplicativos como serviço.
- 

#### 4. Sistemas Distribuídos Pervasivos (Ubíquos)

- **Descrição:** São sistemas compostos por dispositivos de computação muito pequenos ou embutidos no cotidiano, muitas vezes com comunicação via rede sem fio.
  - **Características:**
    - **Mobilidade:** Dispositivos podem ser móveis, com recursos limitados de bateria e processamento.
    - **Conectividade Instável:** As conexões podem ser variáveis, exigindo mecanismos robustos para lidar com interrupções.
    - **Interação com o ambiente:** Usados para computação ubíqua, como em casas inteligentes, wearables ou sensores ambientais.
  - **Exemplo:** Redes de sensores para monitoramento ambiental ou sistemas domésticos inteligentes onde dispositivos interagem para, por exemplo, ajustar a iluminação e a temperatura automaticamente.
- 

#### Resumo Visual (Comparativo):

Modelo	Homogeneidade	Comunicação	Escopo de Aplicação
--------	---------------	-------------	---------------------

<b>Cluster</b>	Alta (mesmo hardware/SO)	Rede local (baixa latência)	Processamento intensivo e paralelo
<b>Grid</b>	Baixa (recursos variados)	Rede ampla (com middleware)	Colaboração científica, grandes problemas
<b>Cloud</b>	Virtualizada	Internet/Sistemas de dados	Serviços sob demanda e escaláveis
<b>Pervasivo</b>	Dispositivos pequenos e heterogêneos	Geralmente wireless	Aplicações do cotidiano e IoT

## ✓ 4. Transparência em Sistemas Distribuídos

---

### 🧠 Definição:

Transparência é a capacidade do sistema distribuído de ocultar dos usuários e dos programadores o fato de que os recursos e os processos estão fisicamente distribuídos.

Em um sistema “transparente”, o usuário não sabe (e não precisa saber) onde está o recurso, se ele foi movido, replicado, se caiu, etc.

---

### 🎯 Objetivo da Transparência:

- Facilitar o uso
  - Reduzir erros humanos
  - Aumentar a robustez e a consistência do sistema
  - Esconder detalhes técnicos e geográficos da infraestrutura
-



## Tipos de Transparência (7 principais)

Tipo	O que oculta?	Exemplo prático
<b>Acesso</b>	Diferença entre acessar recurso local ou remoto	Abrir um arquivo no Google Drive como se fosse local
<b>Localização</b>	A localização física do recurso ou serviço	Usar uma URL sem saber onde o servidor está
<b>Migração</b>	Recurso pode ser movido de um nó a outro sem que o usuário perceba	Mover um banco de dados entre servidores à noite
<b>Realocação</b>	Recurso pode ser movido <b>durante o uso</b>	Stream de vídeo muda de servidor sem travar
<b>Replicação</b>	Existem múltiplas cópias do recurso	Vários servidores web entregando a mesma página
<b>Concorrência</b>	Vários usuários acessando o mesmo recurso ao mesmo tempo, sem conflitos	Dois usuários editando o mesmo documento no Drive
<b>Falha</b>	O sistema lida com falhas de forma oculta, sem impactar o usuário	Sistema redireciona acesso se um servidor cair



### Dica para lembrar:

Você pode usar a palavra “**ALMRRCF**” como mnemônico para lembrar os tipos:

**Acesso – Localização – Migração – Realocação – Replicação – Concorrência – Falha**



### Observação importante:

Transparência **tem custo!**

- Nem sempre é desejável ocultar tudo.
- Por exemplo, ocultar a latência em um sistema entre Brasil e Europa pode prejudicar decisões do usuário (ex: enviar arquivo para uma impressora lenta e distante).

## Questão clássica de prova:

**Pergunta:** Ocultar do usuário que existem várias cópias de um recurso é transparência de:

- (A) Acesso
- (B) Concorrência
- (C) Replicação 
- (D) Migração

---

## 5. Escalabilidade em Sistemas Distribuídos

---

### Definição:

Escalabilidade é a **capacidade de um sistema distribuído continuar eficiente mesmo quando há aumento no número de usuários, volume de dados ou quantidade de tarefas.**

“Um sistema escalável se adapta ao crescimento sem perder desempenho.”

---

### Tipos de Escalabilidade:

	Tipo	O que é?
Horizontal	Adição de <b>mais máquinas</b> ao sistema (ex: mais servidores)	
Vertical	Potencializar uma máquina existente (ex: mais memória ou CPU)	
Geográfica	Supor te a nós distribuídos em diferentes regiões (ex: Google, Netflix)	

---

### Desafios da Escalabilidade:

#### 1. Gargalos (bottlenecks)

- Ex: Um único servidor autenticando todos os usuários.

- Solução: **Replicação de serviços, balanceamento de carga.**

## 2. Pontos únicos de falha

- Ex: Um banco de dados central cair e o sistema todo parar.
- Solução: **Tolerância a falhas, distribuição de serviços.**

## 3. Aumento de latência

- Quanto mais distante o usuário, maior o tempo de resposta.
- Solução: **CDNs, múltiplos data centers, cache.**



### Técnicas para escalar um sistema:

Técnica	Explicação
Distribuição de carga	Repassar requisições para diferentes servidores
Armazenamento em cache	Guardar dados localmente para evitar requisições repetidas
Elasticidade	Adicionar/remover recursos sob demanda (ex: subir VM na AWS)
Sharding	Dividir dados entre servidores para consulta mais rápida

---



### Exemplos práticos:

- **Netflix** usa servidores diferentes no Brasil, EUA, Europa para entregar vídeo com menor latência e sem sobrecarregar um único ponto.
- **Google Docs** atende milhões de usuários simultaneamente sem travar → usa balanceamento, replicação e cache.
- **Sisu** e **ENEM** muitas vezes falham por não escalar corretamente → um único servidor para milhões de acessos.

---

### Dica para prova:

"A escalabilidade está relacionada à **capacidade de manter desempenho eficiente mesmo com aumento de carga.**"

---

### ! Frase clássica que já caiu em questão:

"A adição de novos dispositivos em um sistema distribuído para atender a uma demanda crescente está ligada à **escalabilidade** do sistema."

---

## 6. Processos e Threads

---

### O que é um processo?

Um **processo** é um **programa em execução**, com tudo que é necessário para funcionar:

- Código
- Dados (variáveis)
- Pilha (chamadas de funções, parâmetros)
- Contexto (estado dos registradores, contador de programa, arquivos abertos etc.)

Ex: Abrir dois Chromes = dois **processos** diferentes, mesmo programa.

---

### O que é uma thread?

Uma **thread** (ou “linha de execução”) é uma **subunidade dentro de um processo**.

Um processo pode ter **várias threads** executando partes do seu código ao mesmo tempo, **compartilhando a mesma memória**.

---

## Diferenças entre Processo e Thread:

Aspecto	Processo	Thread
Memória	Cada um tem a sua	Compartilham o mesmo espaço de memória
Custo de criação	Alto (envolve muito contexto)	Baixo (mais leve)
Comunicação	Lenta (via IPC, pipes, sockets etc.)	Rápida (via variáveis compartilhadas)
Isolamento	Sim, um processo não interfere em outro	Não há isolamento entre threads
Exemplo	Dois apps rodando	Múltiplas abas no navegador Chrome

---

## Por que usar threads?

- Melhor desempenho em **sistemas multicore**
  - Melhor uso da **CPU** (ex: enquanto uma thread espera, outra executa)
  - Menor consumo de **memória e tempo**
  - Ideal para tarefas **concorrentes** ou **em segundo plano**
- 

## Componentes de uma Thread:

- Identificador
- Contador de programa (PC)
- Registradores
- Pilha

**Compartilhado com outras threads do mesmo processo:**

- Seção de código
  - Dados
  - Arquivos abertos
- 

## Criação de Processos e Threads:

- **UNIX/Linux:**
  - `fork()` → cria um novo processo (cópia exata do pai)
- **Windows:**
  - `CreateProcess()` → cria novo processo, podendo carregar outro programa
- **Java/Python:**
  - `threading.Thread()` → cria e inicia threads

Em java: Um ExecutorService é uma interface que define métodos para executar tarefas em um pool de threads. ThreadPoolExecutor é uma classe que implementa ExecutorService e oferece maior controle sobre o pool de threads, como o tamanho do pool e a forma como as tarefas são tratadas. Exemplo:

```
import java.util.concurrent.ExecutorService;  
  
import java.util.concurrent.Executors;  
  
public class ThreadPoolExample {  
  
    public static void main(String[] args) {  
  
        // Cria um pool de threads com um número fixo de threads  
  
        ExecutorService executor = Executors.newFixedThreadPool(5);  
  
        // Submete tarefas para execução no pool de threads  
  
        for (int i = 0; i < 10; i++) {
```

```
final int taskNumber = i + 1;

executor.submit(() -> {

    System.out.println("Executando tarefa: " + taskNumber + " por " +
Thread.currentThread().getName());

});

}

// Encerra o ExecutorService

executor.shutdown();

}

}
```

---

### Estados de um Processo:

1. **Pronto** – esperando a CPU
  2. **Executando** – usando a CPU
  3. **Bloqueado** – esperando um evento externo (ex: entrada de dados)
- 

### Desafios com Threads:

- Concorrência e **condições de corrida** (race conditions)
  - Problemas de **sincronização** (acesso ao mesmo recurso ao mesmo tempo)
  - Complexidade na depuração
- 

### Exemplo prático (Python):

```
python
import threading

def tarefa(numero):
    print(f"Thread {numero} executando...")

for i in range(4):
    t = threading.Thread(target=tarefa, args=(i,))
    t.start()
```

---

### 👉 Pergunta típica de prova:

“Threads de diferentes processos compartilham memória?”

Falso. Threads do mesmo processo compartilham memória. Threads de processos diferentes, não.

---

## ✓ 7. Concorrência vs Paralelismo

---

### 🧠 Definições-chave:

Conceito	O que é?
Concorrência	Quando várias tarefas estão em progresso ao mesmo tempo, mas não necessariamente executando ao mesmo tempo
Paralelismo	Quando várias tarefas são executadas exatamente ao mesmo tempo, em diferentes núcleos de CPU

---

### 💡 Analogia simples:

Imagine uma cozinha com várias receitas sendo preparadas:

- **Concorrência:** um único cozinheiro alterna entre as receitas — pica os legumes, depois mexe o molho, depois volta para a carne (troca de contexto).
- **Paralelismo:** vários cozinheiros trabalhando simultaneamente, cada um com sua receita.

---

## Exemplos práticos:

Situação	Concorrente ?	Paralelo?
Um único núcleo alternando entre tarefas	 Sim	 Não
Um processador com 4 núcleos, cada um com uma thread	 Sim	 Sim
Um programa rodando uma tarefa de cada vez	 Não	 Não

---

## Na prática:

- **Concorrência** depende do **escalonamento** feito pelo sistema operacional (ex: um núcleo simula multitarefa).
  - **Paralelismo** exige **hardware com múltiplos núcleos** ou múltiplos processadores.
- 

## Threads podem ser usadas para ambos?

 Sim!

- Você pode criar várias threads mesmo com apenas **um núcleo** (concorrência).
  - Com vários núcleos, threads podem rodar **realmente em paralelo**.
- 

## Cuidados com concorrência:

- Acesso simultâneo à memória compartilhada pode gerar **condições de corrida**.
- Requer **sincronização**: semáforos, locks, mutex etc.

---

### Pergunta comum em prova:

“Threads só funcionam com múltiplos núcleos?”

 **Falso** — threads funcionam com 1 núcleo (concorrência) e também com múltiplos (paralelismo real).

---

## 8. Comunicação entre Processos (IPC)

---

### O que é IPC?

É o conjunto de mecanismos que permite que **processos diferentes troquem dados ou sinais entre si**, seja **no mesmo computador ou em diferentes máquinas**.

---

### Por que IPC é necessário?

- Processos não compartilham memória por padrão.
  - Precisamos de formas para que eles colaborem e coordenem tarefas.
  - Em sistemas distribuídos, a comunicação ocorre **exclusivamente por troca de mensagens**.
- 

### Tipos de Comunicação (local ou remota):

Mecanismo	Descrição
Sinais (signals)	Envia notificações simples entre processos (ex: <code>SIGINT</code> , <code>SIGKILL</code> )
Fluxos padrão	<code>stdin</code> , <code>stdout</code> , <code>stderr</code> ; usados para entrada e saída entre processos

**Pipes (	)**
<b>Sockets</b>	Comunicação entre processos locais ou remotos via rede
<b>Memória Compartilhada</b>	Dois processos acessam a mesma área de memória
<b>Message Queue / Mailbox</b>	Fila de mensagens com envio/recebimento assíncrono

---

## Classificações de Comunicação:

### 1. Síncrona vs Assíncrona

Tipo	Descrição
<b>Síncrona</b>	O processo <b>espera</b> a resposta (bloqueante)
<b>Assíncrona</b>	O processo <b>continua executando</b> após o envio da mensagem

### 2. Persistente vs Transiente

Tipo	Descrição
<b>Persistente</b>	A mensagem é armazenada até o receptor estar disponível
<b>Transiente</b>	A mensagem se perde se o receptor não estiver disponível

---

## Exemplo de combinação (como pode cair na prova):

Comunicação via socket TCP entre cliente e servidor, onde o cliente continua executando após o envio da mensagem, mas o servidor precisa estar online no momento.

## Comunicação Assíncrona + Transiente

---

## O que é um socket?

- É um **ponto de extremidade** para comunicação entre dois processos.

- Pode ser usado tanto para comunicação local quanto entre máquinas remotas (TCP/IP).
- Muito usado em redes e na internet (ex: navegador se comunicando com um site).

```
python
# Exemplo básico em Python
import socket
s = socket.socket()
s.connect(('localhost', 1234))
s.send(b'Hello')
```

---

## Middleware Procedural e Socket

- Um **middleware procedural** (como RPC) **usa sockets por baixo dos panos** para fazer a comunicação entre cliente e servidor.
  - Ele **abstrai** os detalhes de conexão, empacotamento e envio de mensagens.
- 

## Dica para lembrar:

"Sockets são a **base** para comunicação entre processos **remotos**. Middleware como RPC usa sockets para parecer que você está chamando funções locais."

---

## Pergunta de prova:

"O tipo de comunicação em que o processo **remetente não espera** resposta e a mensagem **não é armazenada** caso o destino não esteja disponível é chamada de:"

Assíncrona + Transiente

---

## 9. RPC – Remote Procedure Call

---

### O que é RPC?

**RPC (Chamada de Procedimento Remoto)** é um mecanismo que permite a um programa executar uma função em outro computador da rede **como se fosse local**.

O programador escreve `resultado = somar(5, 7);` e o sistema cuida de enviar essa chamada para um servidor remoto, executar a função, e trazer o resultado.

---

### Por que usar RPC?

- **Abstração:** O programador não precisa lidar diretamente com sockets, pacotes ou conexões.
  - **Produtividade:** Código mais limpo e reutilizável.
  - **Integração distribuída:** Muito usado em sistemas cliente-servidor, microserviços e middleware.
- 

### Como o RPC funciona?

1. O **cliente** chama uma função como se fosse local.
  2. Um **stub do cliente** empacota (marshal) os parâmetros e envia para o servidor.
  3. O **servidor** recebe, desempacota (unmarshal) e executa a função.
  4. O resultado é empacotado novamente e enviado de volta ao cliente.
  5. O stub do cliente recebe e retorna o valor para o programa.
-



## Componentes do RPC:

Componente	Função
<b>Cliente</b>	Programa que faz a chamada remota
<b>Stub do cliente</b>	Empacota os dados e envia pela rede
<b>Servidor</b>	Onde o procedimento real está implementado
<b>Stub do servidor</b>	Recebe os dados, executa e devolve o resultado

---



## Exemplo ilustrativo:

Imagine o cliente chamando:

```
c  
int resultado = somar(3, 4);
```

Essa chamada vai para o servidor, que tem:

```
c  
int somar(int a, int b) {  
    return a + b;  
}
```

O cliente **nem sabe** que essa função foi executada remotamente.

---



## Síncrono ou assíncrono?

- RPC **tradicional** é **síncrono** → o cliente **espera** a resposta.
  - Existem variações com **RPC assíncrono** → o cliente **continua** executando após enviar.
-

## **! Variações:**

- **RMI (Remote Method Invocation)**: RPC orientado a objetos (Java)
  - **gRPC (Google RPC)**: Versão moderna baseada em protocolo HTTP/2 + Protobuf
- 

## **🎓 Questão típica:**

"O mecanismo de RPC é bastante utilizado para comunicação em sistemas distribuídos. O RPC assíncrono permite que:"

- (D) O processo que faz a chamada **possa continuar executando** após o envio da mensagem. 
- 

## **🧠 Dica de prova:**

"RPC mascara a comunicação de rede e faz parecer que estamos chamando funções locais, mesmo sendo executadas remotamente."

## 10. Speedup em Programação Paralela

---

### **🧠 O que é Speedup?**

**Speedup** é uma métrica usada para quantificar o **ganho de desempenho** ao se executar um algoritmo em paralelo, comparado com sua versão sequencial.

"Se com 1 thread leva 10 segundos, e com 4 threads leva 3 segundos, o speedup mede esse ganho."

---

### **Fórmula do Speedup:**

$$\text{Speedup} = \frac{T_s}{T_p}$$

- **T<sub>s</sub>** (**Tempo sequencial**) = tempo de execução da versão com **1 thread/processo**

- $T_p$  (**Tempo paralelo**) = tempo de execução com  $n$  threads/processos
- 

### Exemplo:

Execução	Tempo (s)
Sequencial	10
Paralelo (4 threads)	3
Speedup=10/3≈3,33	

---

### Speedup Ideal vs Real:

	Característica
<b>Ideal</b>	Quando Speedup = número de threads
<b>Sublinear (real)</b>	Normal (perdas por sincronização, overhead)
<b>Superlinear</b>	Raro, pode ocorrer por uso eficiente de cache

---

### Fatores que afetam o Speedup:

1. **Overhead** de criar threads/processos
2. **Sincronização** (locks, semáforos, I/O)
3. **Carga mal distribuída** entre as threads
4. **Acesso à memória compartilhada**
5. **Latência de comunicação** (em sistemas distribuídos)

---

## Coleta de tempos (Benchmarking):

Ao medir tempos para calcular speedup:

- **Warm-up:** descarte as primeiras execuções (a JVM ou cache pode aquecer).
  - **Outliers:** descarte os valores muito altos ou baixos (percentil 5% ou 95%)
  - **Repetição:** execute várias vezes para ter média confiável.
- 

## Boas práticas:

- Testar com **2, 4, 8, 16 threads** e comparar com o sequencial.
  - Usar **múltiplos núcleos** reais (senão é só concorrência, não paralelismo).
  - Avaliar **qual parte do código paraleliza bem**.
- 

## Pergunta típica:

“Se uma tarefa leva 20 segundos no modo sequencial e 5 segundos no modo paralelo com 4 threads, qual o speedup?”

```
\text{Speedup} = \frac{20}{5} = 4  
]
```

---

## Dica para prova:

"Speedup ideal é quando o tempo de execução é reduzido proporcionalmente ao número de threads/processos."

---

Tudo certo até aqui!  
Podemos seguir para o **último tema**?

## ► 11. Desafios em Sistemas Distribuídos?

**Você disse:**

Próximo

**O ChatGPT disse:**

Ótimo, Cayo! Vamos para o **último tema: Desafios em Sistemas Distribuídos**, que ajuda a entender por que esse tipo de sistema exige tanto cuidado no projeto.

---

## ✓ 11. Desafios em Sistemas Distribuídos

---

### 🧠 Visão Geral:

Sistemas distribuídos oferecem muitos benefícios, mas enfrentam **vários desafios** técnicos que afetam desempenho, confiabilidade, segurança e manutenção.

---

### ⚠️ Principais Desafios e Como Lidar com Eles:

Desafio	Por que é um problema?	Estratégias para lidar
<b>Escalabilidade</b>	Aumentar usuários ou recursos pode causar lentidão ou falhas	Balanceamento de carga, cache, arquitetura em camadas
<b>Heterogeneidade</b>	Sistemas com diferentes SOs, redes, hardwares	Middleware, protocolos padrão (ex: REST, gRPC)
<b>Segurança</b>	Comunicação por rede pode ser interceptada, alterada ou invadida	Criptografia, autenticação, firewalls, tokens
<b>Tratamento de falhas</b>	Quedas de rede, servidores ou dados perdidos	Replicação, tolerância a falhas, backups
<b>Concorrência</b>	Acesso simultâneo a recursos pode causar inconsistências	Locks, transações, controle de concorrência

<b>Tempo de resposta</b>	Latência pode ser alta em redes geograficamente distribuídas	CDNs, pré-carregamento, compressão de dados
<b>Sincronização de relógio</b>	Diferença de tempo entre máquinas pode afetar ordenação de eventos	Protocolos como NTP, timestamp lógico (ex: Lamport)
<b>Gerenciamento de recursos</b>	Recursos precisam ser monitorados e alocados de forma eficiente	Monitores, orquestradores, escalonadores de tarefa

---

## Detalhes sobre alguns desafios:

### 1. Concorrência

- Problemas: race conditions, deadlocks
- Solução: mecanismos de sincronização (mutex, semáforos), escalonamento justo

### 2. Tolerância a Falhas

- O sistema deve **continuar funcionando mesmo com falhas parciais.**
- Estratégias:
  - Detecção de falhas (heartbeats)
  - Redundância (replicação de serviços e dados)
  - Failover automático

### 3. Segurança

- Desafios:
  - Proteger dados em trânsito
  - Autenticar quem acessa
- Técnicas:
  - Criptografia (TLS/SSL)

- Autenticação via tokens (ex: JWT)
  - Controle de acesso baseado em papéis
- 

### Dica para prova:

"Sistemas distribuídos devem lidar com falhas de componentes, concorrência de processos e a escalabilidade da rede para garantir robustez."

---

### Pergunta típica:

"A diferença entre os relógios de máquinas distintas em um sistema distribuído pode causar problemas de ordenação de eventos. Como mitigar isso?"

- Usar **timestamps lógicos** (ex: algoritmo de Lamport)
- 

### Resumo rápido para memorização:

- Escalabilidade
- Heterogeneidade
- Segurança
- Falhas
- Concorrência
- Tempo de resposta
- Sincronização de relógio

**Mnemônico:** *EHSFCTS* — parece nome de protocolo maluco, mas ajuda!

---

## Tema 12: Middleware em Sistemas Distribuídos

---

### O que é Middleware?

Middleware é uma **camada intermediária de software** que **facilita a comunicação entre aplicações distribuídas**. Ele fica entre o sistema operacional e os aplicativos.

“Pense no middleware como o **encanamento invisível** que conecta sistemas diferentes, escondendo detalhes como sockets, pacotes, IPs e protocolos.”

---

### Objetivos do Middleware:

- Abstrair a complexidade da rede
  - Facilitar a comunicação entre processos remotos
  - Aumentar a portabilidade e interoperabilidade
  - Gerenciar conexões, mensagens, transações, sessões, segurança
- 

### Categorias de Middleware:

Tipo	Descrição
Middleware Procedural	Comunicação baseada em <b>funções e procedimentos</b>
Middleware Orientado a Objetos	Comunicação via <b>invocação de métodos em objetos remotos</b>
Middleware Orientado a Mensagens	Baseado em <b>troca de mensagens assíncronas</b> (ex: Kafka, RabbitMQ)
Middleware Transacional	Suporte a <b>transações distribuídas</b> , com rollback automático



## Middleware Procedural: RPC (Remote Procedure Call)

Já vimos este!

- O cliente chama um procedimento como se fosse local, mas ele é executado em outro processo, possivelmente remoto.
- O middleware cuida da **serialização, envio, resposta e desserialização** dos dados.

Exemplo: `resultado = somar(2, 3);` – mas essa função está no servidor.

**Características:**

- Mais fácil de usar que sockets
  - Pode ser **síncrono ou assíncrono**
  - Usado em C, C++, Java, Python (via `xmlrpc, grpc`)
- 



## Middleware Orientado a Objetos: RMI (Remote Method Invocation)

- RMI é a **versão orientada a objetos do RPC**, usada especialmente em Java.
- Permite invocar **métodos de objetos remotos** como se estivessem locais.

Exemplo:

```
java
MeuObjetoRemoto obj = (MeuObjetoRemoto)
Naming.lookup("//servidor/objeto");
obj.metodoRemoto(); // parece local, mas roda remotamente!
```

**Componentes:**

- Interface remota (`extends java.rmi.Remote`)
- Implementação

- Registro RMI (RMI Registry)
  - Cliente que consome o objeto remoto
- 

## Resumo Comparativo

Middleware	Paradigma	Exemplo de uso
<b>Procedural (RPC)</b>	Chamada de função	<code>resultado = somar(x, y)</code>
<b>Orientado a Objetos (RMI)</b>	Chamada de método OO	<code>obj.metodoRemoto()</code> (em Java)

---

### Vantagens do uso de Middleware:

- Reduz o **acoplamento** entre aplicações
  - Torna sistemas **mais modulares e escaláveis**
  - Permite **interoperabilidade** entre linguagens, sistemas e plataformas
- 

### Dica para a prova:

“Middleware procedural abstrai a chamada de funções remotas; middleware orientado a objetos permite invocação remota de métodos em objetos.”

---

### Questão típica:

“Qual das opções a seguir descreve corretamente o papel de um middleware procedural?”

(A) Gerencia chamadas de procedimentos remotos entre processos e esconde detalhes da comunicação com sockets. 

NOTA: 11

BOA PROVA!

1. (0.6 ponto) Observe os códigos abaixo de um sistema cliente-servidor:

```
from xmlrpclib import SimpleXMLRPCServer
import os
from gtts import gTTS

server = SimpleXMLRPCServer(("192.168.2.107", 9050))
server.register_introspection_functions()

def transformaTextoEmAudio(ptexto):
    tts = gTTS(text=ptexto, lang='pt')
    tts.save("audioaula.mp3")
    os.system("mpg321 audioaula.mp3")
    os.system('start audioaula.mp3')
    return 1

server.register_function(transformaTextoEmAudio)
server.serve_forever()
```

– Código do servidor.

```
import xmlrpclib
s = xmlrpclib.ServerProxy('http://192.168.2.107:9050')
s.transformaTextoEmAudio("Na aula de hoje tivemos prova.")
```

– Código do cliente.

Explique o que este sistema faz. Além disso, classifique-o quanto ao tipo de comunicação (transiente, persistente, síncrono e assíncrono). Em caso de síncrono, determine o nível de sincronismo (para envio, entrega ou processamento de requisição). Justifique sua classificação. *Não fiz!*

2. (0.6 ponto) Alguns dos principais desafios gerais de Sistemas Distribuídos são: escalabilidade, heterogeneidade, segurança, tratamento de falhas e concorrência. No sistema acima, você consegue identificar e explicar 3 destes desafios não tratados? Descreva possíveis estratégias que usaria para solucioná-los.

3. (0.7 ponto) O que são processos e o que são threads? Em sua resposta, diga (1) quem monitora estes algoritmos em execução e (2) cite no mínimo 3 itens que os diferencie em nível de projeto.

4. (0.6 ponto) Considere o resumo do estudo “Analizando a Escalabilidade e a Acurácia de Implementações Paralelas e Distribuídas para a Detecção de Comunidades em Grafos” (SSCAD, 2023):

“Detecção de comunidades em grafos é um tipo de análise amplamente utilizada por aplicações de diversas áreas do conhecimento. Com o crescente aumento do volume de dados surge a necessidade de implementações paralelas e distribuídas para que o tempo de processamento não aumente a ponto de limitar sua aplicabilidade. Este estudo analisa a escalabilidade e a acurácia dos métodos para detecção de comunidades mais utilizados atualmente em ambientes paralelos e distribuídos. A partir de uma análise detalhada do impacto que a comunicação entre os processos teve no tempo de execução de cada implementação são feitas recomendações em relação aos algoritmos mais promissores em relação a melhoria da escalabilidade nestes ambientes.”

O objetivo deste estudo é propor algoritmos mais promissores do que os mais utilizados atualmente para detecção de comunidades em grafos. Você, enquanto um cientista em final de formação, mesmo não sendo conhecedor profundo de Teoria dos Grafos, sabe alguns conceitos relacionados à otimização de algoritmos paralelos e distribuídos. Assim, diga no mínimo 4 itens relacionados à metodologia de avaliação/resultados e análise que você acha relevante encontrar neste artigo. Por que estes itens são importantes? *Não fiz!*

ALUNO:

VALOR: 2,5 PONTOS

NOTA:

1. (0,5 ponto) Observe o código prova-mpi.py abaixo:

```
numpy
mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.get_rank()
nprocos = comm.get_size()

randNum = numpy.zeros(1)
vetorRandNum = numpy.zeros(nprocos)

rank == 0:
    print("Sorteia um número aleatório")
    print("Considerando que o número sorteado foi 0,77")
    randNum = numpy.random.sample(1)

randNum = comm.broadcast(randNum, root = 0)

if rank != 0:
    print("Processo ", rank, " - Número ", randNum)
    randNum = randNum * rank

comm.gatherv(vetorRandNum, randNum, root = 0)

if rank == 0:
    print(vetorRandNum)
```

Supondo que o número sorteado para *randNum* seja 0.77, diga quais as mensagens serão impressas na tela para a seguinte execução em linha de comando no Windows:

mpiexec /np 8 python prova-mpi.py

2. (0,5 ponto) Julgue as alternativas abaixo e marque a alternativa correta:

( ) No geral, RPC é útil apenas em aplicações nas quais são aceitáveis chamadas mal-sucedidas ocasionais.

( ) Tanto RPC quanto RMI fornecem transparência às chamadas locais e remotas em aplicações

distribuídas.

( ) RMI, em sua versão original, permite expressividade nos valores passados por parâmetro, sendo atraente para parâmetros grandes e complexos.

( ) Em RMI, um objeto distribuído pode ser acessado e repassado em qualquer máquina pertencente ao sistema distribuído por referência exclusiva.

a) V F V F

b) F F F F

c) F V V F

d) V F F V

e) V V V V

3. (0,5 ponto) Em se falando de segurança, por que é um aspecto crítico em sistemas distribuídos? Dê no mínimo dois exemplos de agravantes de segurança pelo sistema ser distribuído.

4. (0,5 ponto) O que é criptografia e para que serve? Qual a diferença entre chave pública e chave privada?

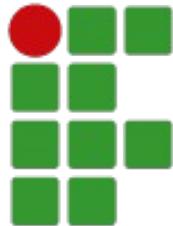
5. (0,5 ponto) Tempo é um parâmetro importante em sistemas distribuídos. Quais os problemas de se usá-lo? Descreva em detalhes um método de sincronização de relógios e diga a que cenário ele melhor se aplica.

## DISCIPLINA SPD – REVISÃO PARA COMUNICAÇÃO EM SD

- 1) O que são sistemas distribuídos? Dê um exemplo prático de sistema distribuído discutindo sobre os seus pontos particulares de comunicabilidade das entidades e de mapeamento na infraestrutura distribuída física.
- 2) Alguns dos principais desafios gerais de Sistemas Distribuídos são: escalabilidade, heterogeneidade, segurança, tratamento de falhas, concorrência e tempo de resposta. Desses fatores, selecione 3 deles. Explique porque são considerados desafios e apresente estratégias que solucionem ou minimizem problemas relacionados a eles.
- 3) O que é transparência em sistemas distribuídos. Dê 3 exemplos, mas discorra sobre eles.
- 4) Em uma rede de sobreposição estruturada, mensagens são roteadas de acordo com a topologia da sobreposição. Cite uma importante desvantagem dessa abordagem e explique como ela pode afetar o sistema.
- 5) Quanto a processos e threads:
  - a) Defina o que é um processo em sistemas operacionais. Como ele difere de um programa?
  - b) Qual a diferença entre processo e thread? Em quais situações é melhor utilizar um ou outro?
  - c) Quem monitora esses algoritmos em execução?
  - d) Explique a diferença entre paralelismo e concorrência. Threads podem ser usadas para ambos?
  - e) Por que a comunicação entre processos (IPC) é importante? Cite pelo menos duas técnicas usadas para isso.
- 6) Volte aos seus dados de experimentação e descreva:
  - a) O que significa *speedup* em contextos de programação paralela? Como ele é calculado?
  - b) Por que é importante considerar *warm-ups*, *outliers* e *overheads* ao coletar dados de desempenho?
  - c) Como o número de threads influencia o desempenho de uma aplicação? Existe um ponto ótimo?
- 7) Sobre os tipos de comunicação, explique:
  - a) As diferenças entre comunicação síncrona e assíncrona.
  - b) A diferença entre comunicação persistente e transitória.
  - c) Dê um exemplo que se encaixe em dois tipos de comunicação citados (a e b) e explique o porquê.
  - d) Quais são as principais diferenças entre comunicação local e comunicação remota entre processos?
  - e) O que é um socket? Como ele é usado para comunicação entre processos remotos?
  - f) Como um middleware procedural se relaciona com um socket?
- 8) Explique o conceito de RPC (Remote Procedure Call). Quais são os principais componentes de um sistema RPC (stub, cliente e servidor)? Como ele facilita a comunicação entre processos em sistemas distribuídos?
- 9) (Pos2010) O mecanismo de RPC é bastante utilizado para a programação em sistemas distribuídos. Implementações mais eficientes desse mecanismo permitem a realização de RPC assíncrono, em que
  - a) o processo que faz a chamada pode fazer uso de redes assíncronas.
  - b) o processo que recebe a chamada pode responder várias chamadas simultaneamente.
  - c) o processo que recebe a chamada responde chamadas assíncronas com máxima prioridade.
  - d) o processo que faz a chamada pode continuar executando após receber confirmação da transmissão.
  - e) os dois processos bloqueiam os demais, a fim de ter acesso exclusivo ao meio de comunicação.

10) O que significa a sigla RPC?

- a) Remote Program Control.
- b) Remote Process Call.
- c) Remote Procedure Call.
- d) Remote Protocol Communication.



**INSTITUTO FEDERAL**  
Sudeste de Minas Gerais



# Sistemas Paralelos e Distribuídos

## Introdução

Bianca Portes

[bianca.castro@ifsudestemg.edu.br](mailto:bianca.castro@ifsudestemg.edu.br)

# Processamento e a Lei de Moore

- Surgiu em 1965, criado por Gordon Earl Moore, co-fundador da Intel

Lei original: a cada 18 meses, o número de transistores nos chips **aumentariam em 100%, mantendo os custos** ao consumidor

Vantagens? Problemas?

# Lei de Moore

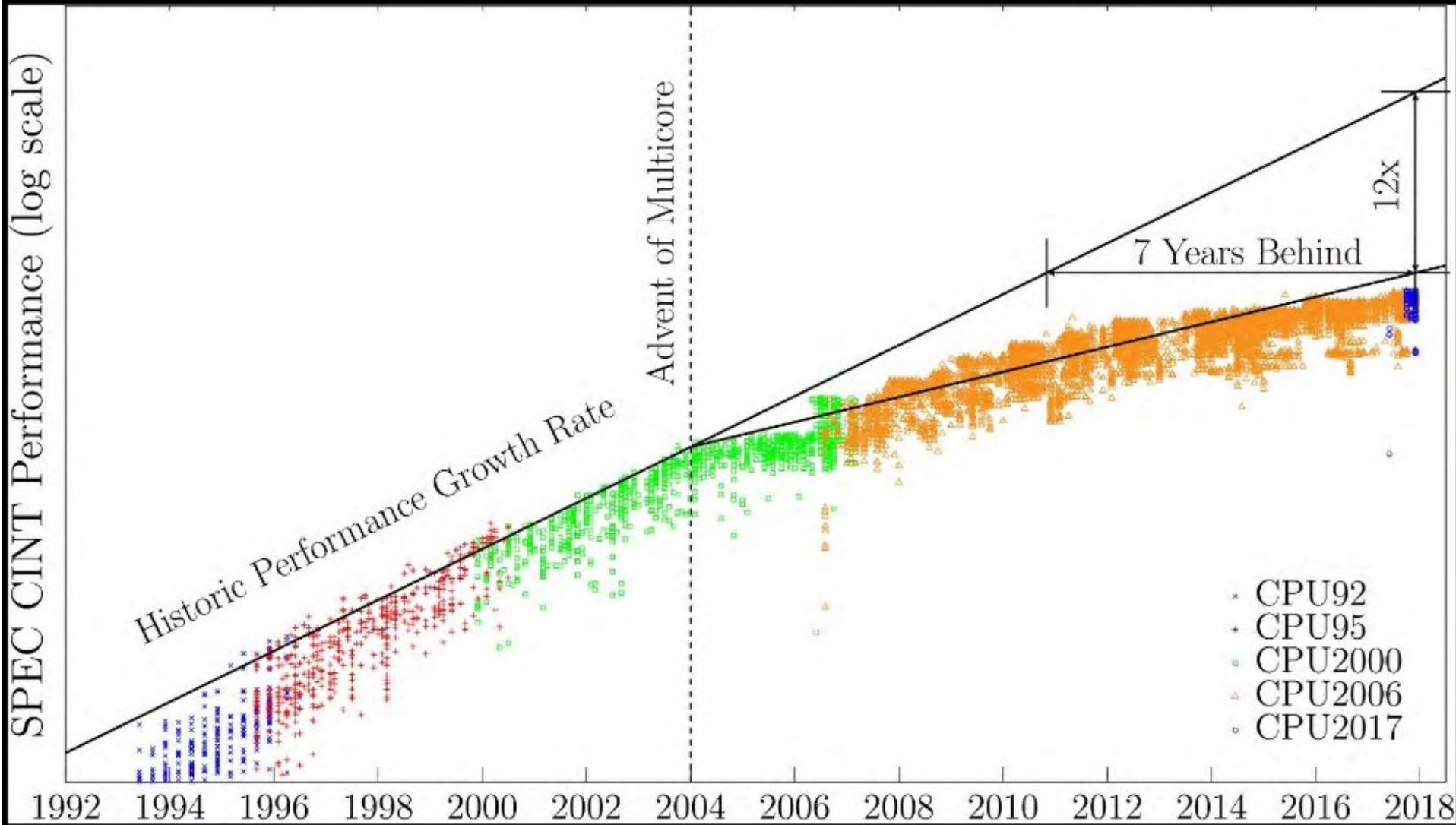
## Vantagens:

- Impulsionou o mercado ano a ano a dobrar sua velocidade, mantendo custos
- Atendia à crescente necessidade por poder computacional do mercado

## Problemas:

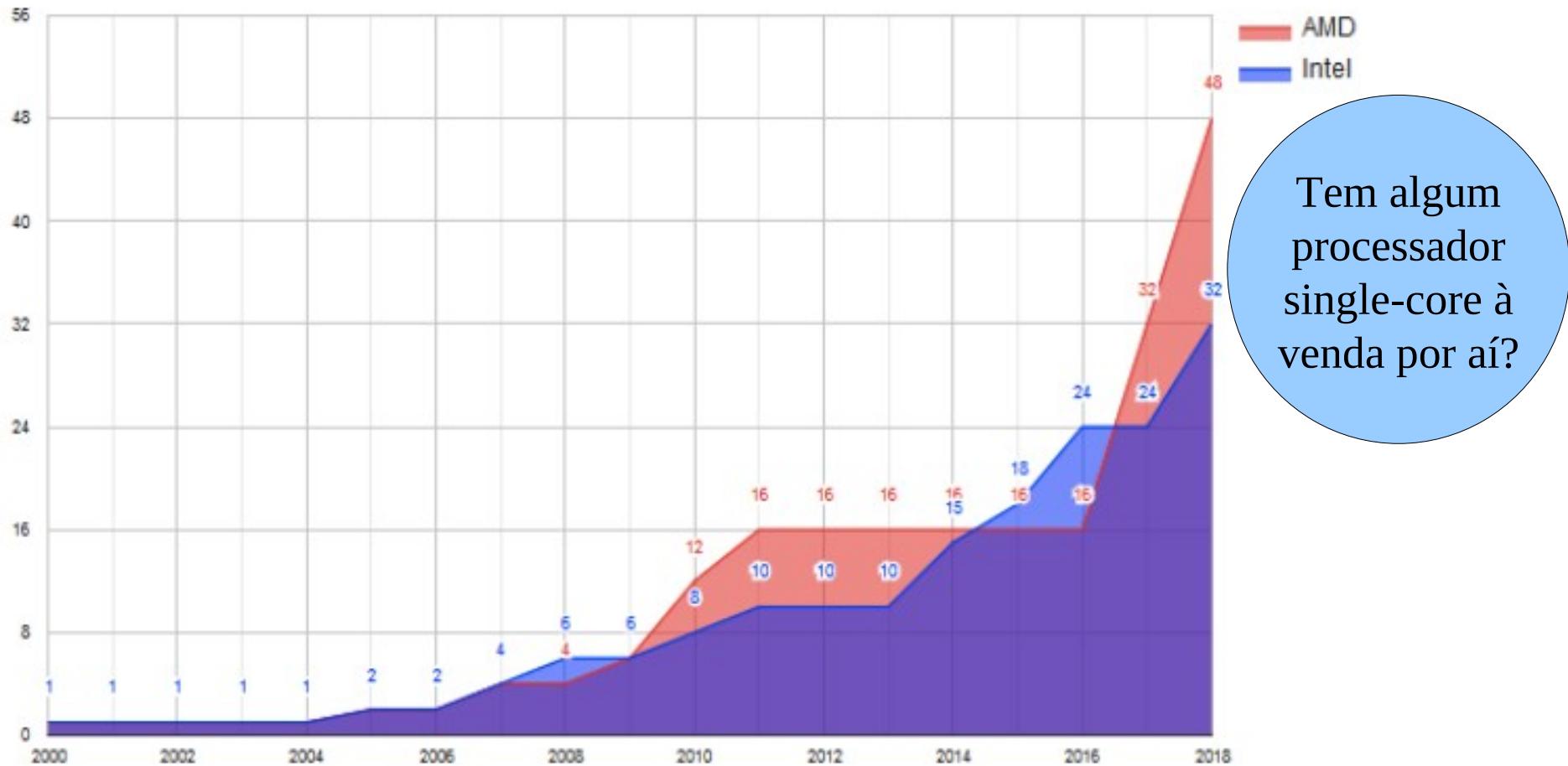
- Muito caro
- Tecnicamente difícil

# A Lei de Moore



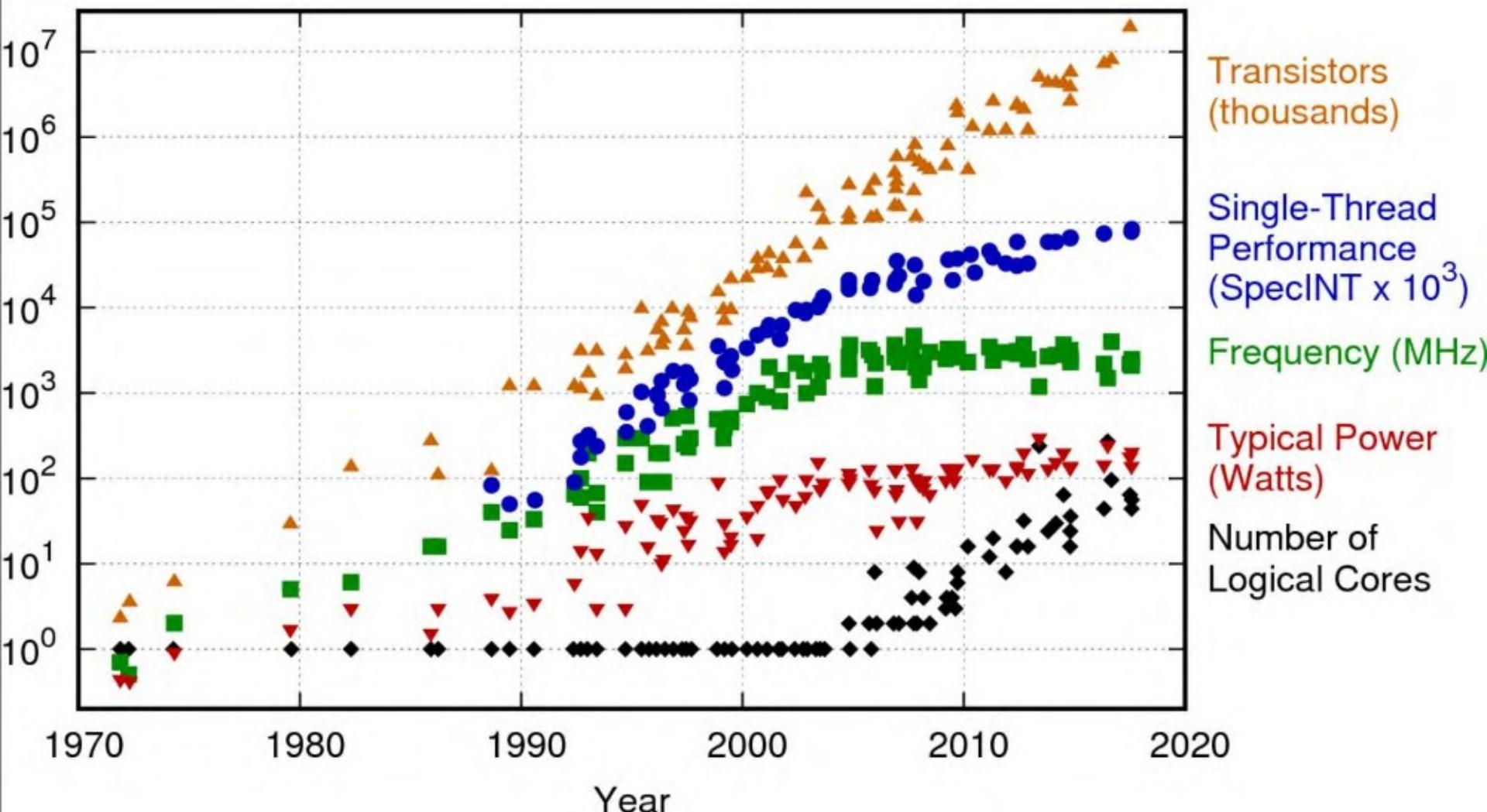
# Evolução da Intel e AMD

Maior quantidade de *cores* por CPU (AMD vs Intel ano a ano)



Tem algum processador single-core à venda por aí?

# Mas a lei de Moore ainda não morreu completamente



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

<https://www.servethehome.com/amd-and-intel-2p-server-core-count-growth-2010-2022/>

# Requisitos de software continuam mudando... o que fazer?



# Programação sequencial?

- Continuar usando modelos de programação sequencial e desperdiçar unidades de processamento já disponíveis?
- Os programas já não são rápidos o suficiente?
- As máquinas já não são rápidas o suficiente?

Como usar essas unidades de processamento de forma simultânea? 9

# Programação Paralela

- Dois dos principais motivos para utilizar programação paralela são:
  - **Reducir o tempo** necessário para solucionar um problema
  - **Resolver problemas mais complexos** e de maior dimensão

# Programação Paralela

- Dois dos principais motivos para utilizar programação paralela são:
  - **Reducir o tempo** necessário para solucionar um problema
  - **Resolver problemas mais complexos** e de maior dimensão

**E a computação distribuída???**

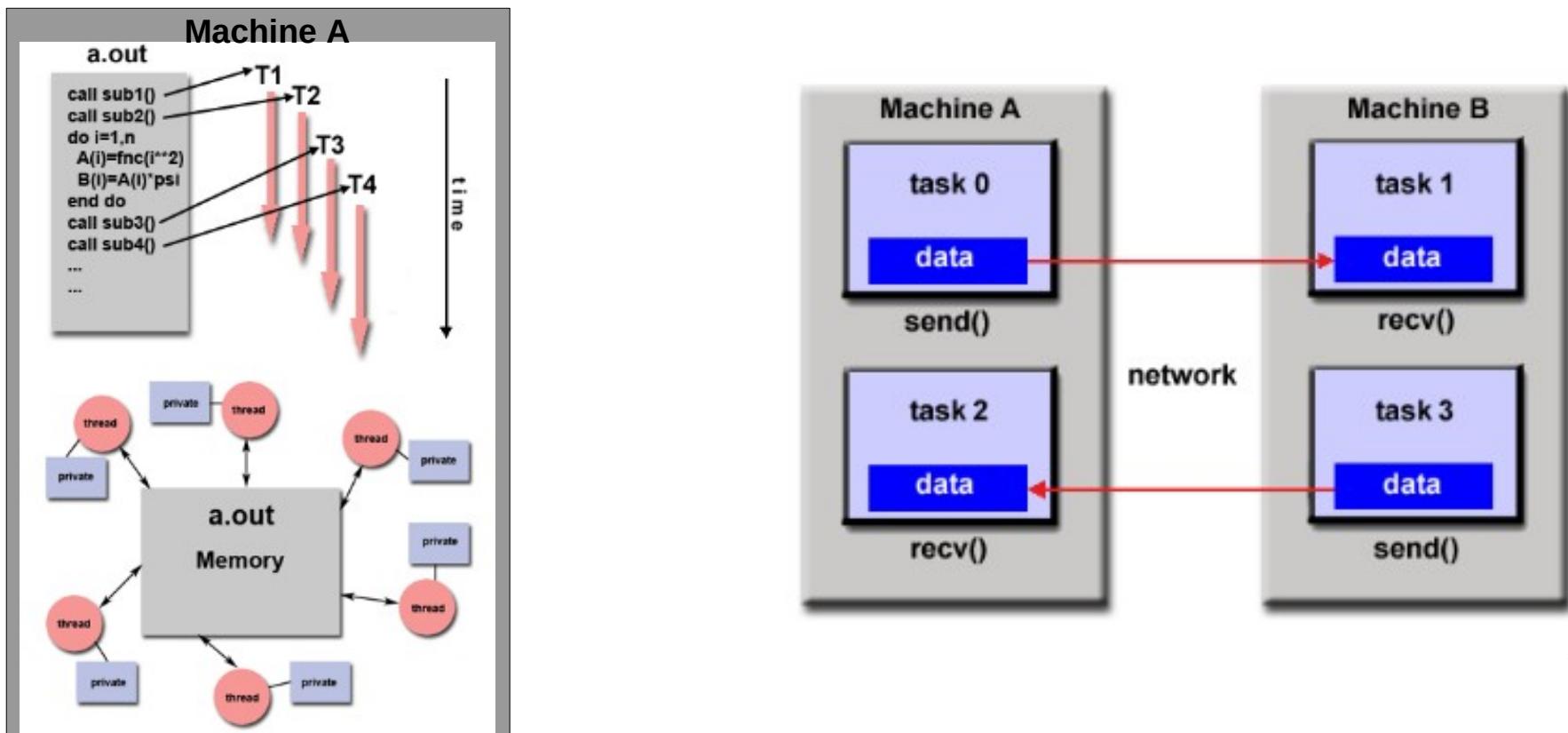
# O que há em comum entre esses modelos de computação?

- Não é incomum usar o termo **computação paralela** para se referir ao mesmo tempo a modelos de **parallelização e distribuição**
  - Buscam extrair **mais desempenho** na computação ao usar os recursos disponíveis através da **computação simultânea**

# SPDs: muito difícil distinguir

Uma das possibilidades:

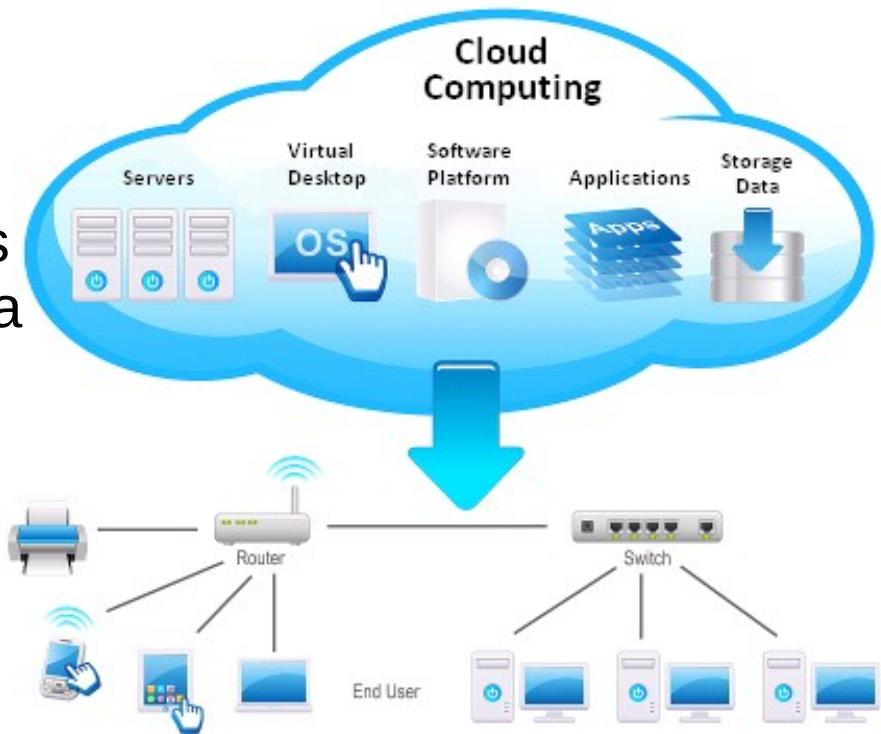
- **paralelização** → modelos associados à **memória compartilhada**
- **distribuição** → a modelos associados à **troca de mensagens**



# Sistemas Distribuídos

- Há vários cenários em que vemos um **sistema distribuído**:
  - Computadores conectados por meio de uma rede e separados por qualquer distância
- Observe que em todos os cenários, estes componentes se apresentam como um único sistema para os usuários (**transparência**)

- **Transparência:** ocultar como esses componentes realmente fazem parte do sistema como um todo



Quem sabe me dizer quando o Gmail já saiu do ar? Ou quando há manutenção? Backup? Quando houveram avarias, como HD queimado?

# Metas em SDs

- **Metas importantes na construção de um sistema distribuído**
  - Fácil acesso a recursos
  - Permitir expansão
  - Transparência (ocultação)

# Metas em SDs

- **Fácil acesso a recursos**
  - Promover acesso a recursos remotos e compartilhamento
    - Ex.: impressoras, processamento, armazenamento, páginas WEB
  - Objetivo: economia/desempenho



# Metas em SDs

- **Expansão (Escalabilidade)**
- Um sistema é escalável quando permanece eficiente mesmo após aumento do número de recursos e de usuários
  - Meta mais importante para desenvolvedores de sistemas distribuídos
- Principais desafios
  - Controlar o custo dos recursos físicos
  - Controlar perdas de desempenho
  - Evitar “gargalos” de desempenho na rede ou nos próprios servidores

# Escalabilidade

- Deve-se tomar cuidado com serviços e/ou dados centralizados quando há possibilidade de uma grande quantidade de usuários

Serviços centralizados	Um único servidor para todos os usuários
Dados centralizados	Sisu, SUS, CGNA
Algoritmos centralizados	Multiplicação de cadeias de matrizes de maneira sequencial

- São possíveis pontos de gargalos → pontos únicos de falhas saturam a rede onde residem

# Metas em SDs

- **Transparência**
  - Um sistema distribuído que é capaz de se apresentar para o usuário como um único sistema é denominado **transparente**
    - Ocultar o fato de que processos e recursos estão fisicamente distribuídos por vários computadores
  - Existem alguns tipos de transparências
    - Acesso
    - Localização
    - Migração
    - Realocação
    - Replicação
    - Concorrência
    - Falha

- Transparência de acesso: permite acesso a componentes remotos e locais através das mesmas operações

Ex.: Google Drive



Google Drive

- Transparência de localização: permite acesso a componentes sem conhecimento da sua localização física
  - Existe um mecanismo que determina a localização baseada em um nome

Ex.: URL

Oculta diferenças na representação de dados e no modo de acesso a um recurso

- Transparência de acesso: permite acesso a componentes remotos e locais através das mesmas operações

Ex.: Google Drive



Google Drive

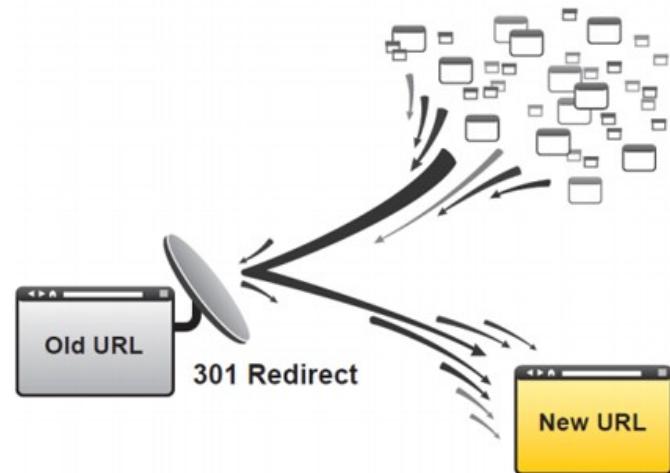
Oculta o lugar onde o recurso está localizado

- Transparência de localização: permite acesso a componentes sem conhecimento da sua localização física

- Existe um mecanismo que determina a localização baseada em um nome

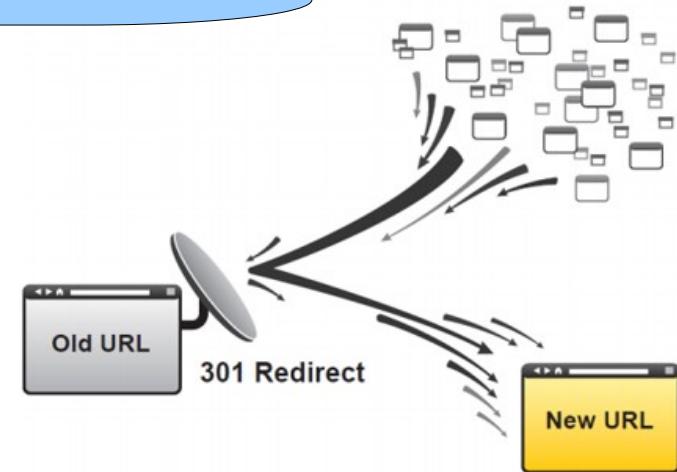
Ex.: URL

- Transparência de migração: um recurso é movido de um local para outro sem que o usuário precise estar ciente. Normalmente, movido de maneira planejada e controlada, como migração de dados entre servidores ou migração de máquinas virtuais entre hosts.
- Transparência de realocação: um recurso é movido mesmo em uso e sem que o usuário perceba a mudança, como uma conexão de rede ou uma aplicação em execução.



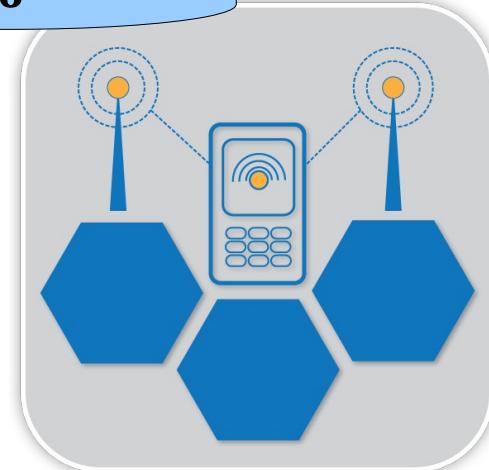
Oculta que um recurso pode ser movido para outra localização

- Transparência de migração: um recurso é movido de um local para outro sem que o usuário precise estar ciente. Normalmente, movido de maneira planejada e controlada, como migração de dados entre servidores ou migração de máquinas virtuais entre hosts.

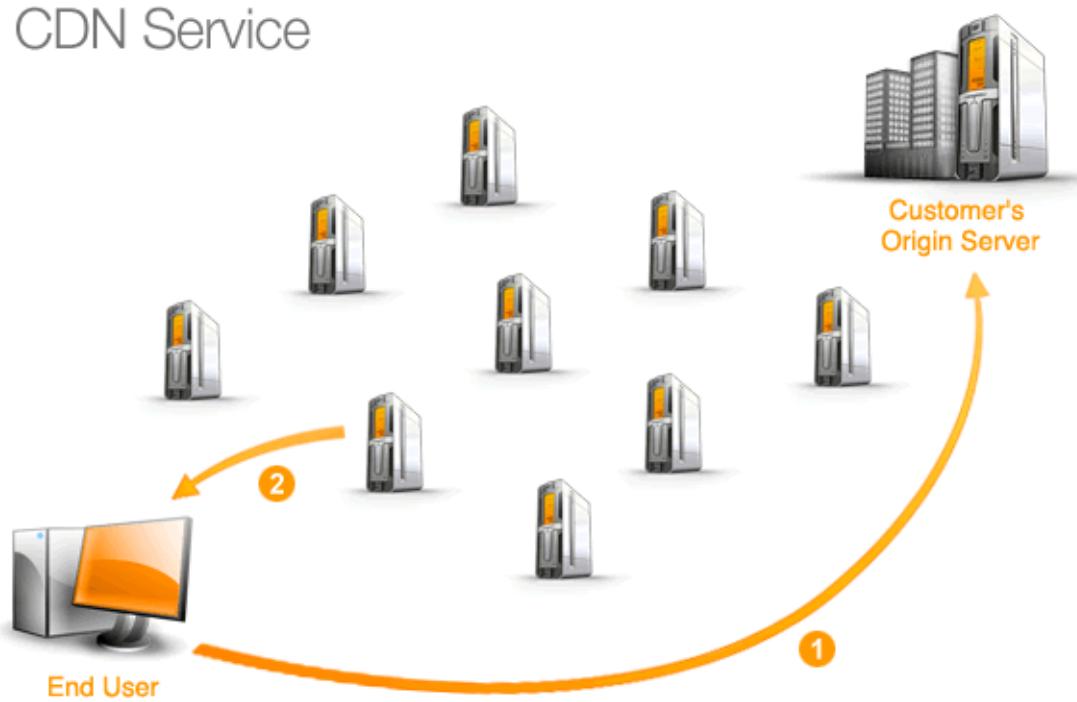


Oculta que um recurso pode ser movido para outra localização **enquanto ele está sendo usado**

- Transparência de realocação: um recurso é movido mesmo em uso e sem que o usuário perceba a mudança, como uma conexão de rede ou uma aplicação em execução.

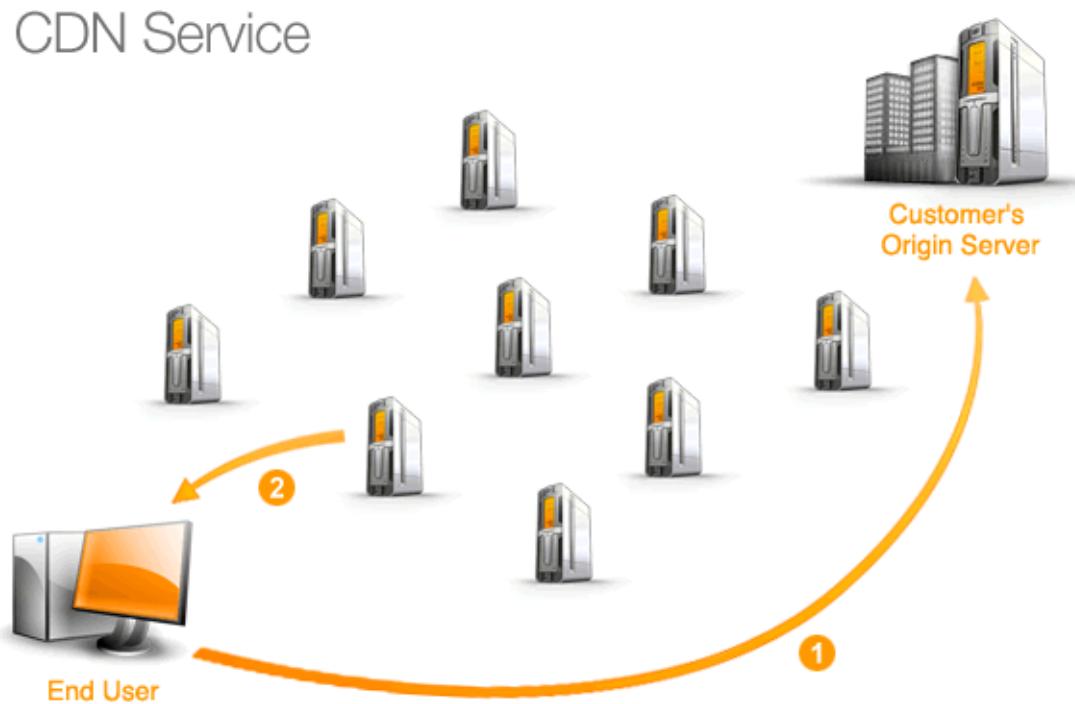


- Transparência de replicação: o usuário desconhece a existência de várias cópias do recurso

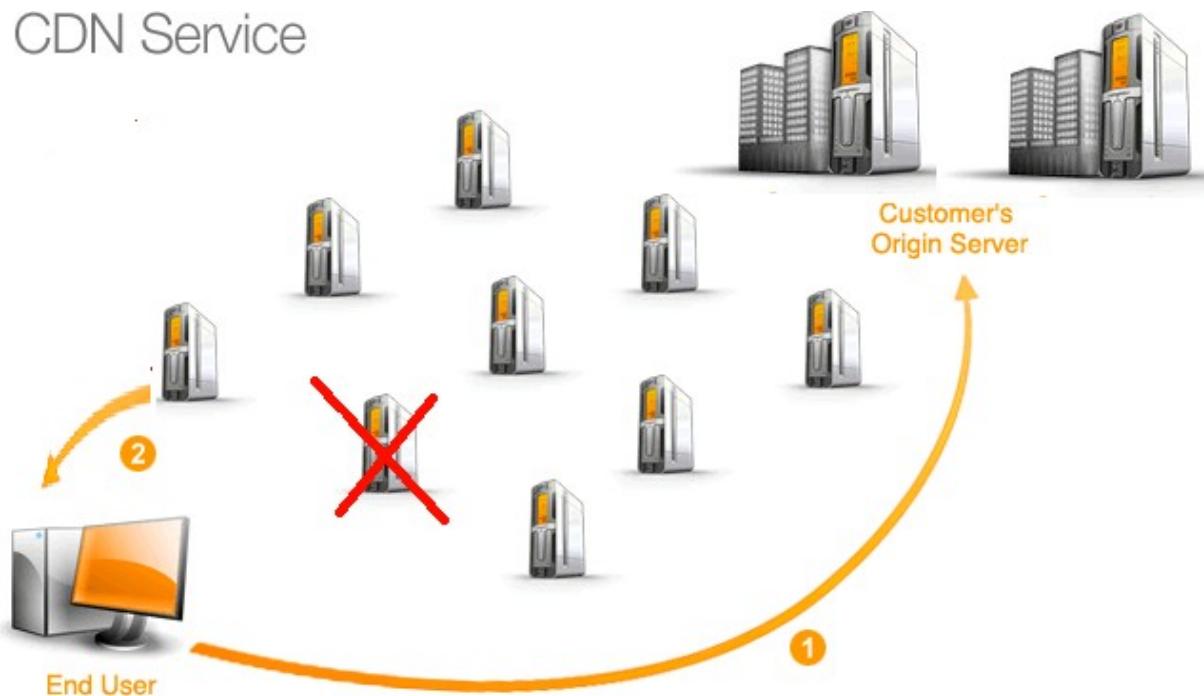


Oculta que um recurso é replicado.

- Transparência de replicação: o usuário desconhece a existência de várias cópias do recurso



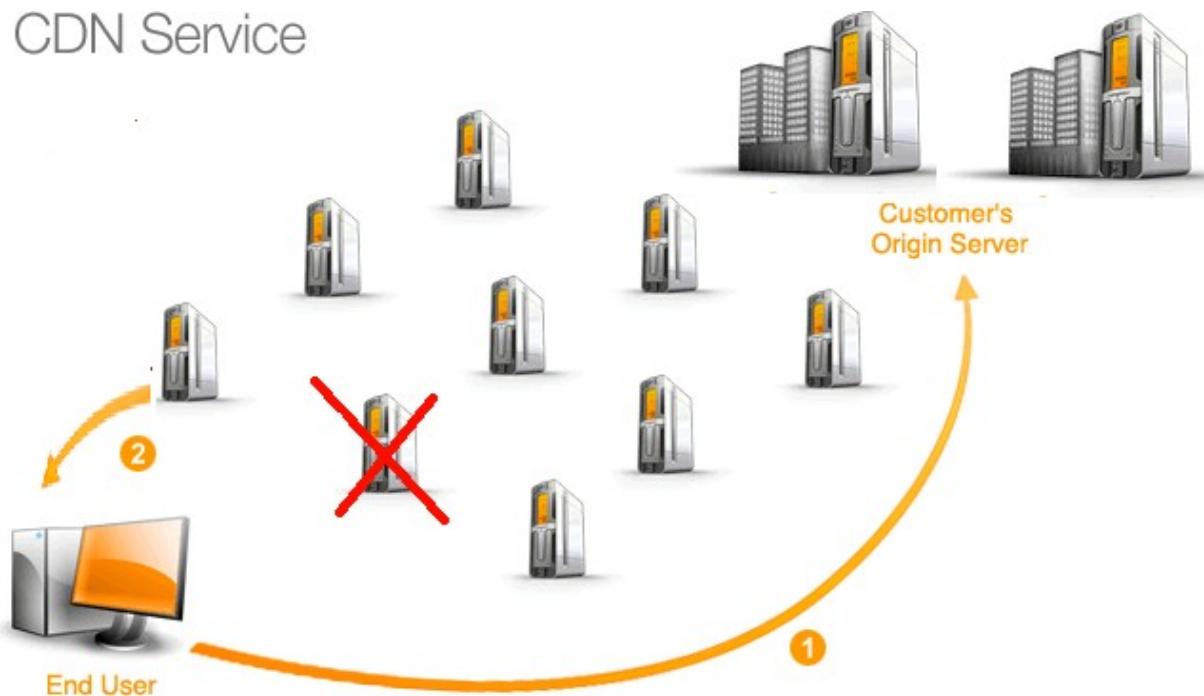
- Transparência de falha: a presença de falhas não é percebida pelos usuários



Oculta a falha e a recuperação de um recurso.

- Transparência de falha: a presença de falhas não é percebida pelos usuários

CDN Service

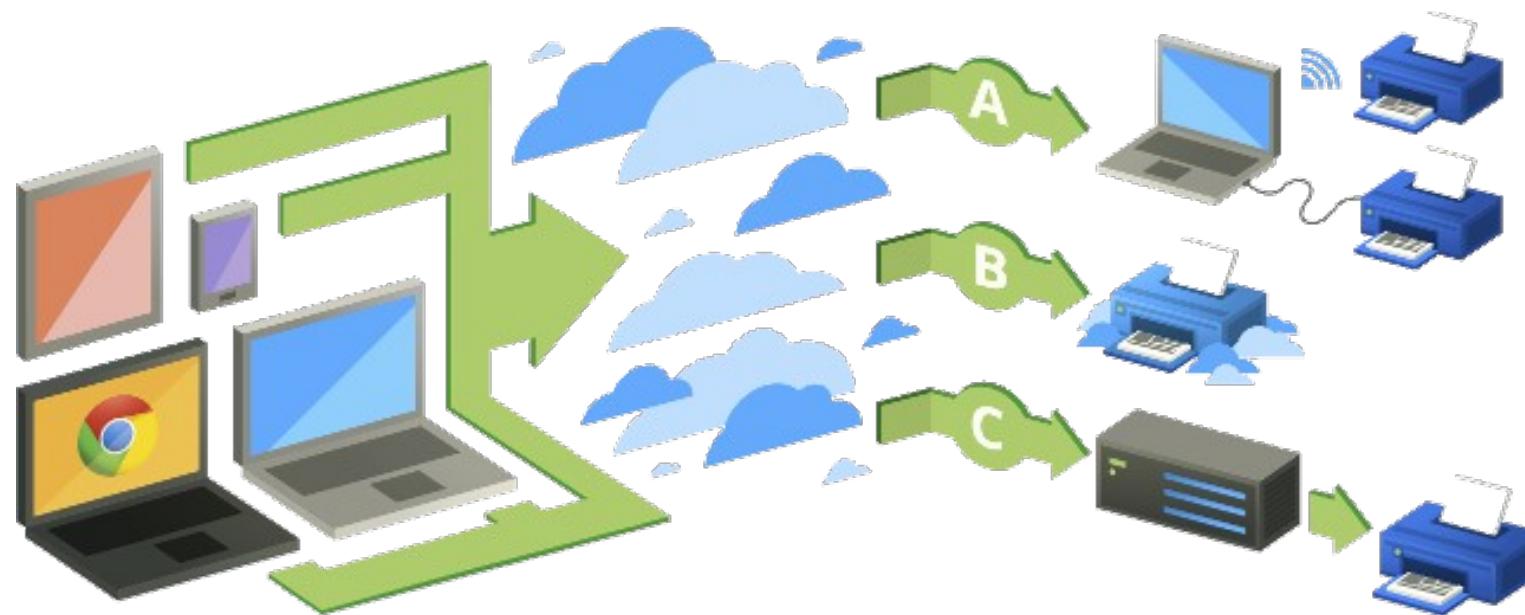


- Transparência de concorrência: permitir que recursos compartilhados sejam acessados de forma segura e consistente, evitando condições de corrida, inconsistências nos dados e outros problemas relacionados à concorrência.



Oculta que um recurso pode ser compartilhado por diversos usuários

- Transparência de concorrência: permitir que recursos compartilhados sejam acessados de forma segura e consistente, evitando condições de corrida, inconsistências nos dados e outros problemas relacionados à concorrência.



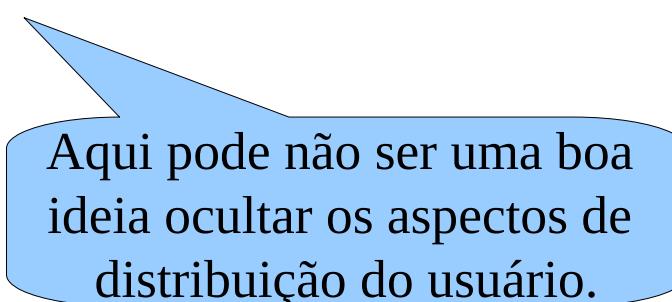
# Metas em SDs

- **Transparência**

- Um sistema é dito transparente para o usuário se ele não perceber que a transparência existe
- Vimos que existem diferentes tipos de transparência em SDs...  
Porém, uma questão mais importante é: a transparência é sempre requerida? Sempre vale “pagar” por esta característica?  
- Ocupação de memória física  
- Fisicamente distribuídos por vários computadores
- Existem alguns tipos de transparências
  - Acesso
  - Localização
  - Migração
  - Relocação
  - Replicação
  - Concorrência
  - Falha

# Exemplos

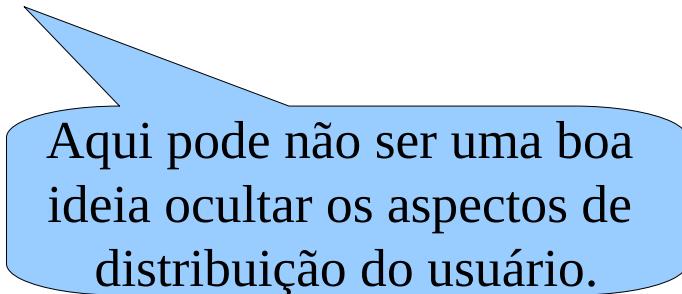
- Em um SD de longa distância que conecta processos entre São Paulo e Amsterdã, por exemplo, não é possível enviar mensagens em menos de 35 ms
  - Achou muito? Dependendo da rede e da capacidade de processamento dos componentes intermediários, pode demorar centenas de milissegundos ou mais



Aqui pode não ser uma boa ideia ocultar os aspectos de distribuição do usuário.

# Exemplos

- Crescimento do uso de dispositivos móveis tornam a noção de localização e contexto cada vez mais importante
  - Exemplo, para uma pessoa, é melhor enviar o trabalho para a impressora ocupada mais próxima ou para a desocupada em um país diferente?



Aqui pode não ser uma boa ideia ocultar os aspectos de distribuição do usuário.

# Questão Poscomp 2015

(Pos2015) Um dos objetivos do projeto de um Sistema Distribuído é fornecer transparência, ocultando aspectos distribuídos dos usuários do sistema. Um sistema transparente proporciona um ambiente em que os seus componentes apresentam-se logicamente centralizados, mesmo fisicamente separados. Entre os vários tipos de transparência que os sistemas distribuídos podem fornecer, o ocultamento do fato de que há várias cópias de um recurso disponíveis no sistema é conhecido como

- (A) transparência de acesso.
- (B) transparência de transação.
- (C) transparência de replicação.
- (D) transparência de concorrência
- (E) transparência de migração.

# Questão Poscomp 2017

Em sistemas de arquivos distribuídos, o requisito no qual os programas clientes não devem conhecer a distribuição de arquivos, sendo que um único conjunto de operações é fornecido para acesso a arquivos locais e remotos, é denominado transparência de

- A) acesso.
- B) desempenho.
- C) escala.
- D) localização.
- E) mobilidade.

# Concurso

(2010, SEPLAG-DF, analista de sistemas) Sistemas distribuídos são aqueles dispostos em computadores distintos e autônomos, que trabalham juntos para dar a impressão de ser um único sistema, com acesso local a seus usuários. Uma característica importante nesses sistemas é a transparência. Acerca das formas de transparência de um sistema distribuído, assinale a alternativa correta.

- A) A transparência de acesso oculta diferenças na apresentação de dados e no modo de acesso a um recurso.
- B) A transparência de replicação oculta que um recurso pode ser movido para outro computador quando em uso.
- C) A transparência de migração oculta que um recurso pode ser compartilhado por diversos usuários, ao mesmo tempo.
- D) A transparência de relocação está presente no sistema quando a falha e a recuperação de um recurso são feitas de forma imperceptível ao seu usuário.
- E) A transparência de localização permite que um recurso possa ser movido de uma localização para outra.

# Concurso

(2020, Prefeitura de Vila Velha, analista de infraestrutura) Em Sistemas Distribuídos, o conceito de transparência pode ser aplicado em vários aspectos. O tipo de transparência que esconde do usuário as diferenças da representação de dados e no modo como um recurso é acessado é chamado de:

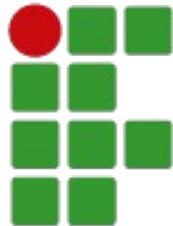
- A) transparência de localização.
- B) transparência de migração.
- C) transparência de relocação.
- D) transparência de acesso.
- E) transparência de concorrência.

# Concurso

(2022, FAPESE, analista) O conceito de transparência pode ser aplicado de diferentes aspectos de sistemas distribuídos.

Assinale a alternativa que indica corretamente o conceito de transparência que esconde diferenças na representação dos dados e a forma pela qual recursos podem ser acessados pelo usuário.

- A) Transparência de Acesso.
- B) Transparência de Aplicação.
- C) Transparência de Relocação.
- D) Transparência de Localização.
- E) Transparência de Apresentação.



**INSTITUTO FEDERAL**  
Sudeste de Minas Gerais



# Sistemas Paralelos e Distribuídos

## Tipos de Sistemas Distribuídos

Bianca Portes

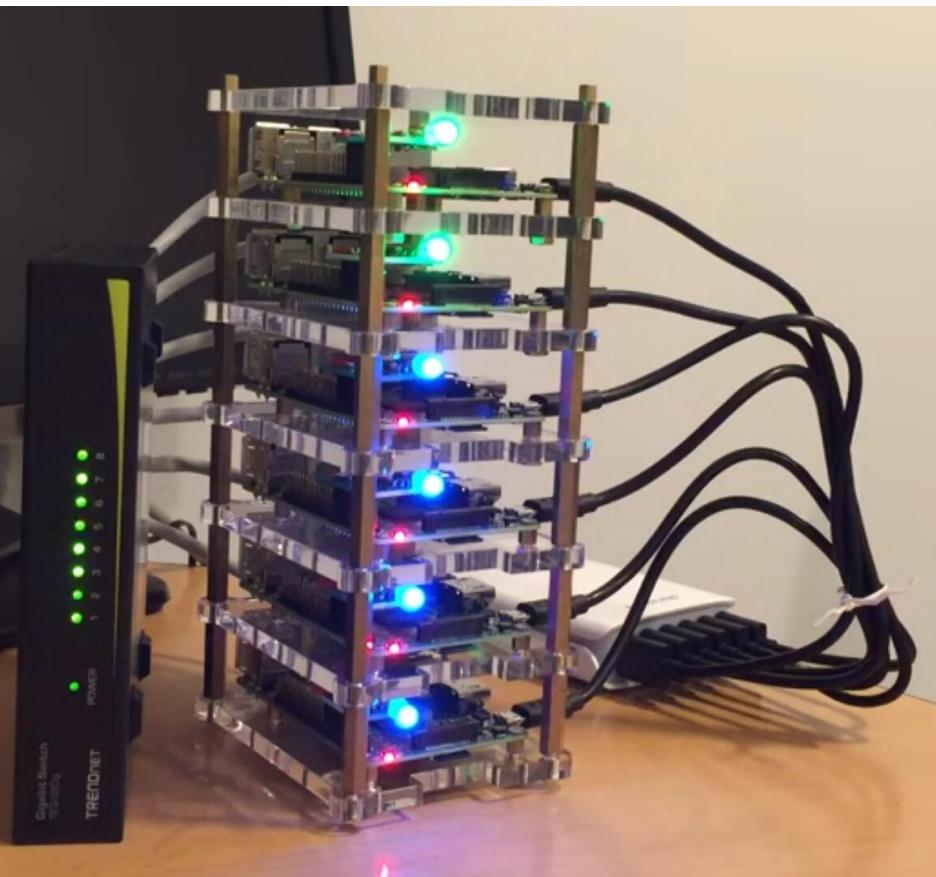
[bianca.castro@ifsudestemg.edu.br](mailto:bianca.castro@ifsudestemg.edu.br)

# Tipos de sistemas distribuídos

- Cluster
- Grid
- Pervasivo

# Tipos de sistemas distribuídos

- **Sistemas de computação de cluster**
  - Um cluster utiliza vários sistemas de computador que são idênticos em hardware e sistema operacional
  - Tornaram-se populares quando a razão preço/desempenho de computadores pessoais e estações de trabalho melhorou



# Tipos de sistemas distribuídos

- **Sistemas de computação em grid (grade)**
  - Sistemas de computador heterogêneos: podem ser diferentes o hardware, sistemas operacionais, redes, domínios administrativos, políticas de segurança e assim por diante



# Tipos de sistemas distribuídos

Os sistemas distribuídos que discutimos até aqui:

- caracterizados por sua estabilidade
  - nós fixos
  - conexão de alta qualidade com uma rede

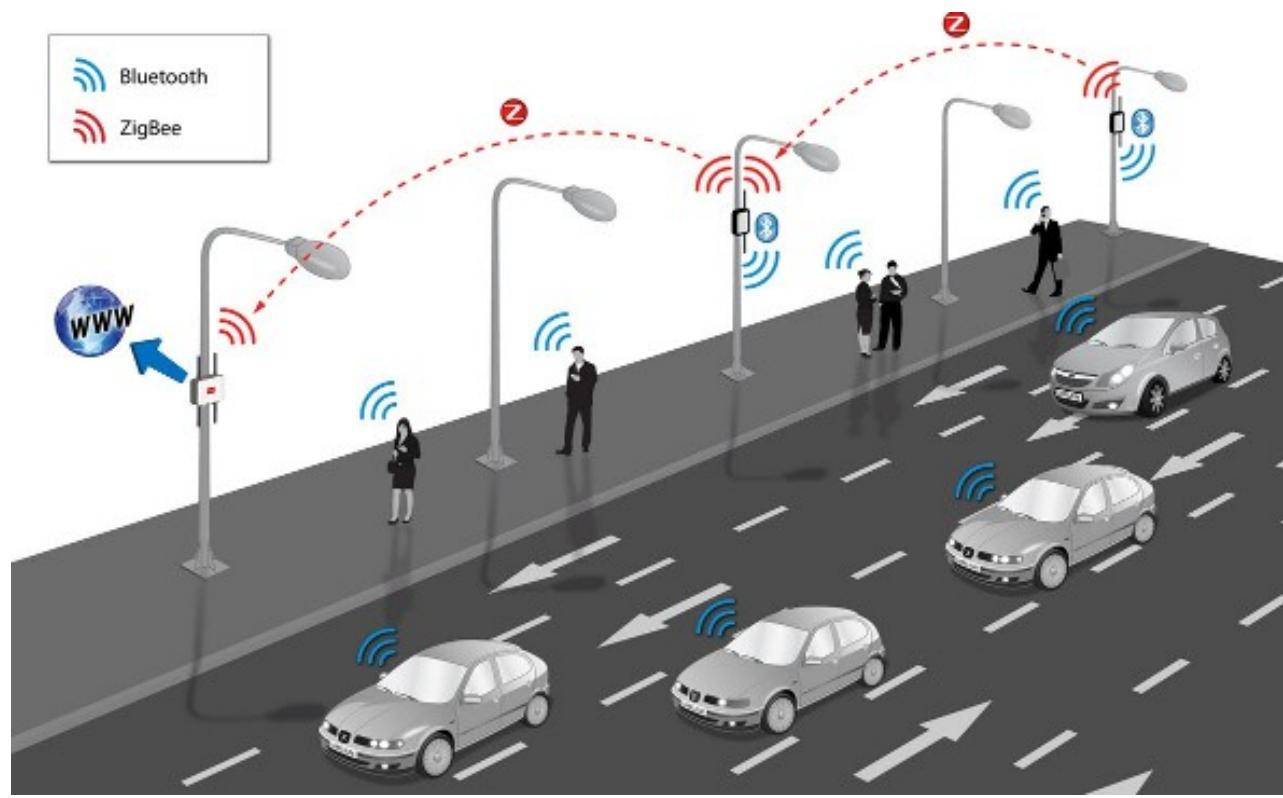
Mas nem só de estabilidade vivem os Sds...

# Tipos de sistemas distribuídos

- **Sistemas distribuídos pervasivos**
  - **Computação pervasiva** também é denominada **computação ubíqua**
  - Aspectos importantes
    - Pequenos equipamentos de computação como objetos diários que mal são notados
      - ausência geral de controle administrativo humano
    - os dispositivos se juntam ao sistema para acessar/fornecer informações
      - Requer meios para ler, armazenar, gerenciar e compartilhar informação com facilidade

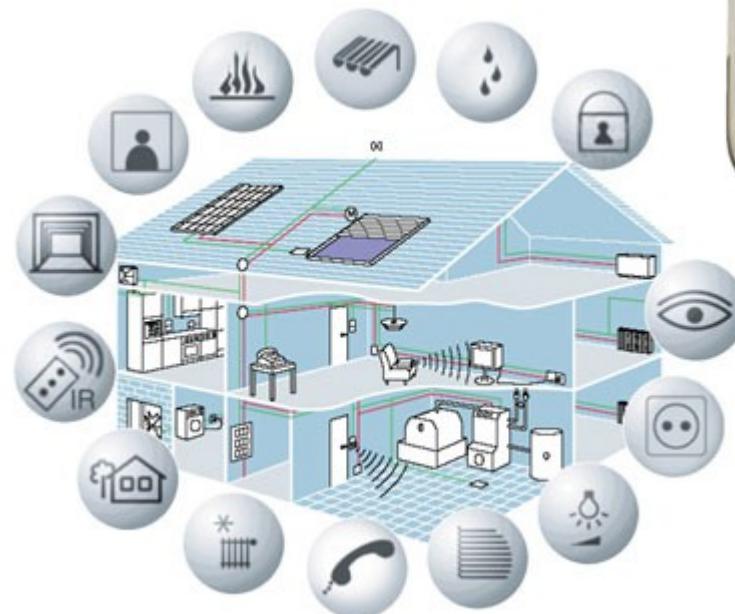
# Tipos de sistemas distribuídos

- Sistemas distribuídos pervasivos
  - Redes de Sensores:



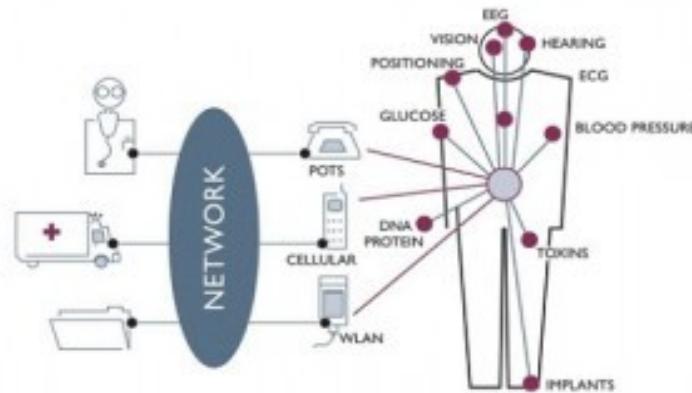
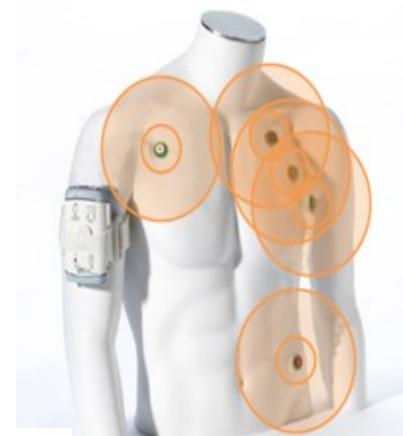
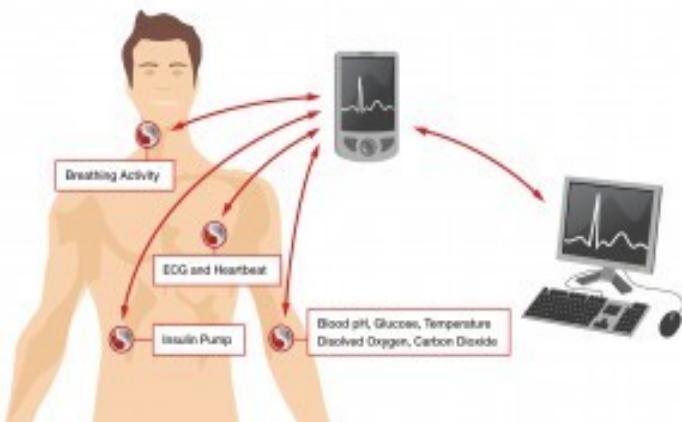
# Tipos de sistemas distribuídos

- Sistemas distribuídos pervasivos
  - Sistemas domésticos:



# Tipos de sistemas distribuídos

- Sistemas distribuídos pervasivos
  - Sistemas eletrônicos para tratamento de saúde:



# Vamos praticar!!

- 1) Explique o que quer dizer transparência (de distribuição) e dê exemplos de diferentes tipos de transparência.
- 2) Explique como funcionaria um sistema escalável.
- 3) A troca de mensagens é um dos principais fatores que afetam o desempenho de um sistema distribuído:

$$\text{Tempo de execução} = \text{Tempo de processamento} + \text{Tempo de comunicação}$$

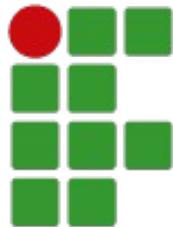
Pesquise as velocidades de comunicação dos seguintes padrões. Indique e justifique o tipo de SD mais indicado para o seu uso:

- |               |              |           |
|---------------|--------------|-----------|
| a) Ethernet   | c) Wi-Fi     | e) ZigBee |
| b) InfiniBand | d) Bluetooth | f) Fibra  |

- 4) Com o máximo de detalhes que souber, esboce um projeto para um sistema multimídia em uma Smart Home. O sistema é composto de um servidor de mídia e um cliente sem fio.

O cliente é quem possui o equipamento (análogo) de áudio e vídeo.

O servidor não possui teclado ou monitor conectados a ele.



**INSTITUTO FEDERAL**  
Sudeste de Minas Gerais



# Sistemas Distribuídos

## Processos

Bianca Portes

[bianca.castro@ifsudestemg.edu.br](mailto:bianca.castro@ifsudestemg.edu.br)

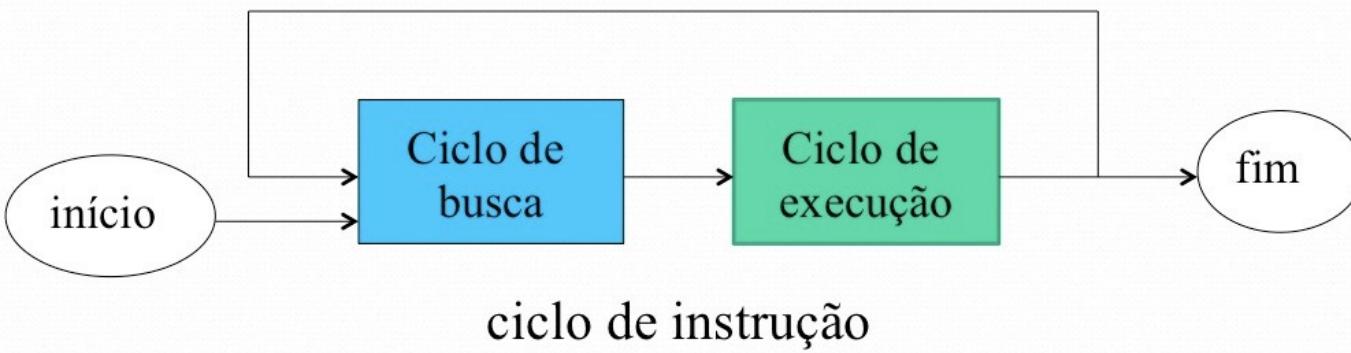
# Retrospectiva

- Até agora:
  - Vimos o que são ou como podem ser os sistemas paralelos/distribuídos
    - Citamos vários exemplos
  - Vimos vários conceitos chaves
    - Escalabilidade, tolerância a falhas, lei de Moore, etc

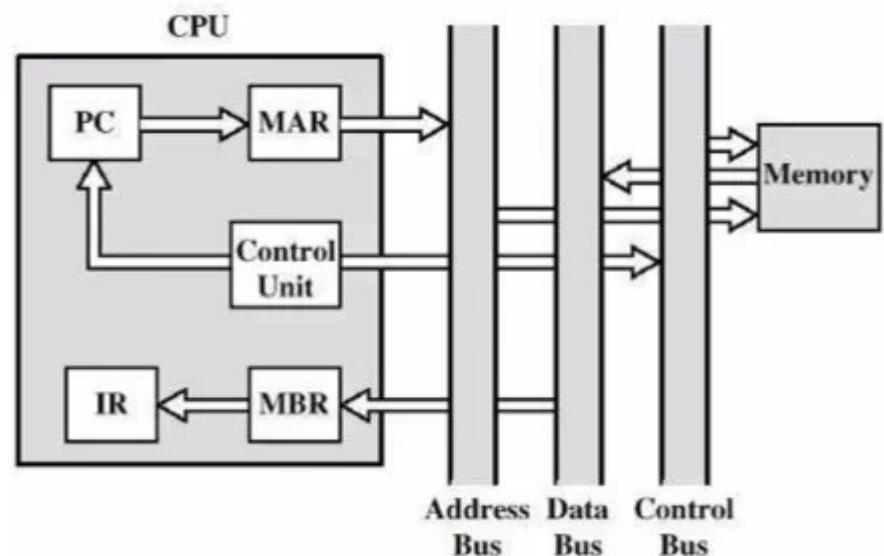
# Nessa aula

- Pretendemos responder as seguintes perguntas
  - O que é um processo?
  - Qual a importância dos processos?

# Estrutura do processamento



ciclo de busca



MBR = Memory buffer register

MAR = Memory address register

IR = Instruction register

PC = Program counter

## - Processos

- Conceito originado de sistemas operacionais
  - Em geral, são definidos como **programas em execução**
- O gerenciamento de processos é a questão mais importantes discutidas em Sistemas Operacionais (SO)

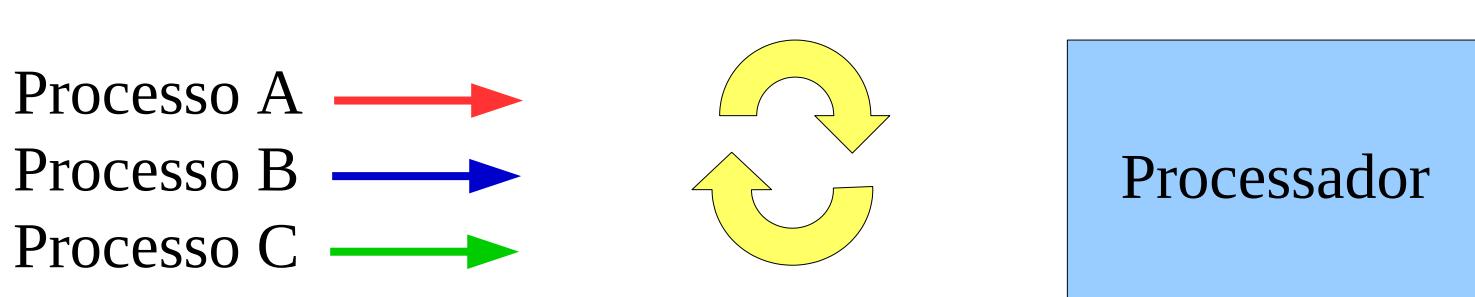


- Todo S.O. fornece interfaces para que os usuários tenham algum controle sobre os programas em execução

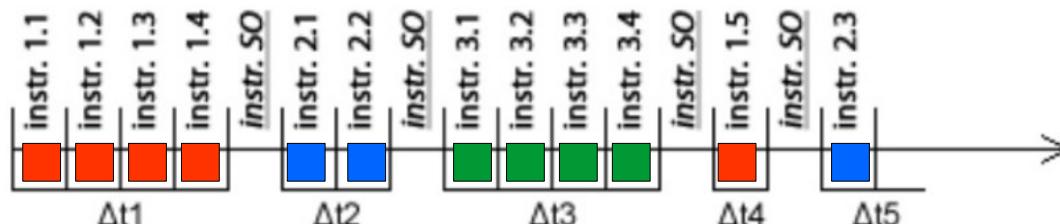
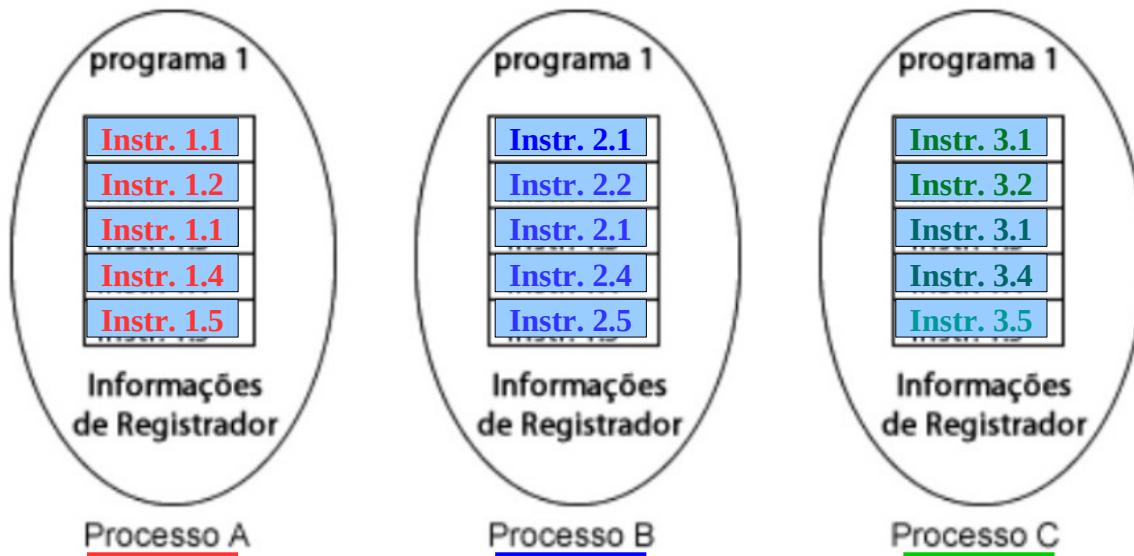
Consegue citar alguma?

## - Processos

- **Aspecto importante:** o SO toma grande cuidado para assegurar que processos independentes não afetem, de modo intencional ou não, o comportamento um do outro
- Os processos podem compartilhar concorrentemente a mesma CPU e outros recursos de hardware de forma transparente



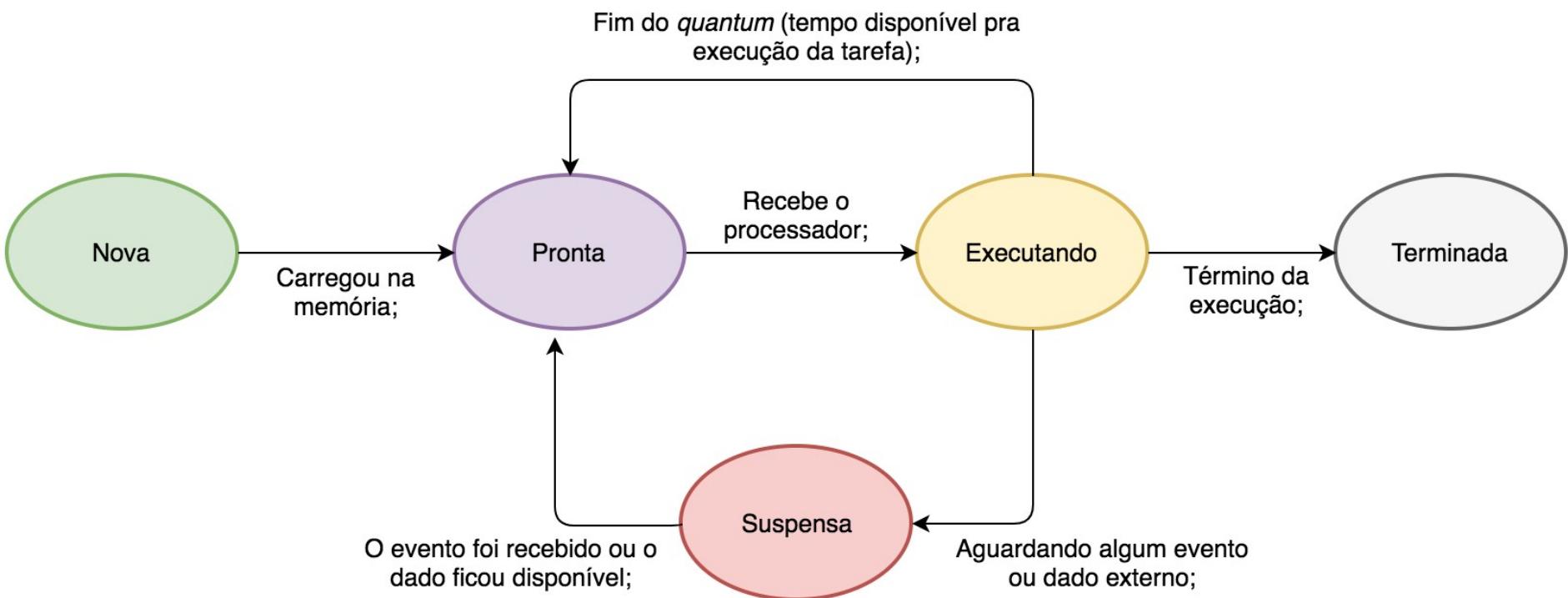
# - Processos



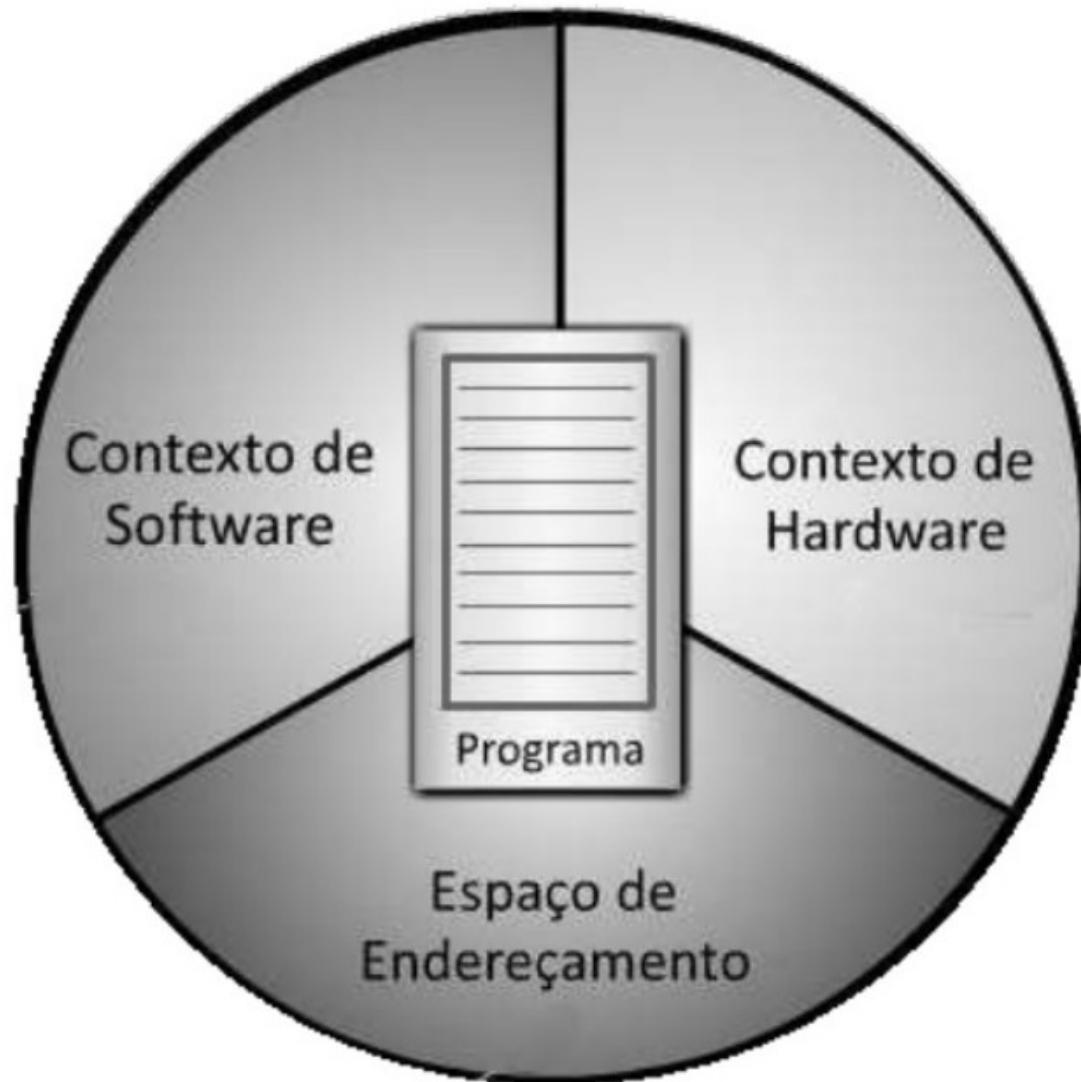
- Essa transparência tem preço relativamente alto... lembra por quê?

# - Processos

- Cada processo possui um **contexto**
  - **Contexto** é uma coleção de dados sobre cada processo que permite que o processador volte a executar processos suspensos ou interrompidos



# Valores que acompanham o processo



# Contexto de hardware

- Referente ao conteúdo dos registradores
- É fundamental para a alternância dos processos na utilização da CPU
  - Mudança de contexto
    - Salva o conteúdo dos registradores da CPU de um determinado processo
    - Carrega os novos valores referentes ao processo que está ganhando a utilização do processador

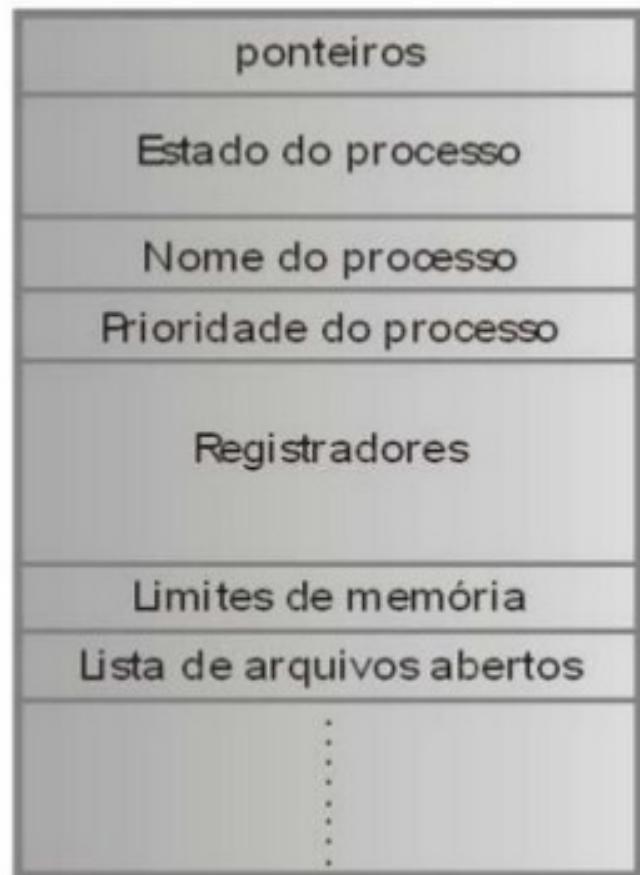


# Contexto de software

- A maior parte das informações de software do processo provém de um arquivo do SO: arquivo de usuários
- O contexto de software é composto por 3 grupos de informações sobre o processo:
  - Identificação
  - Quotas
  - Privilégios

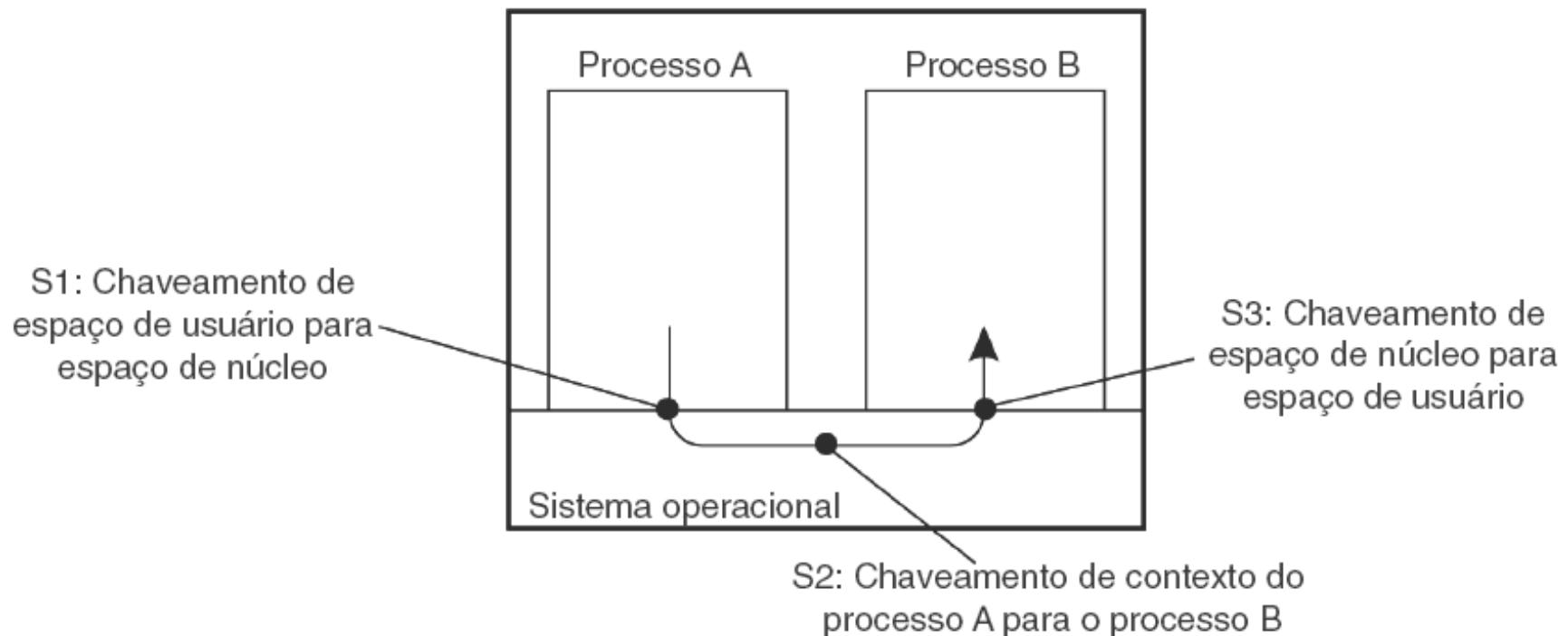
# Espaço de endereçamento

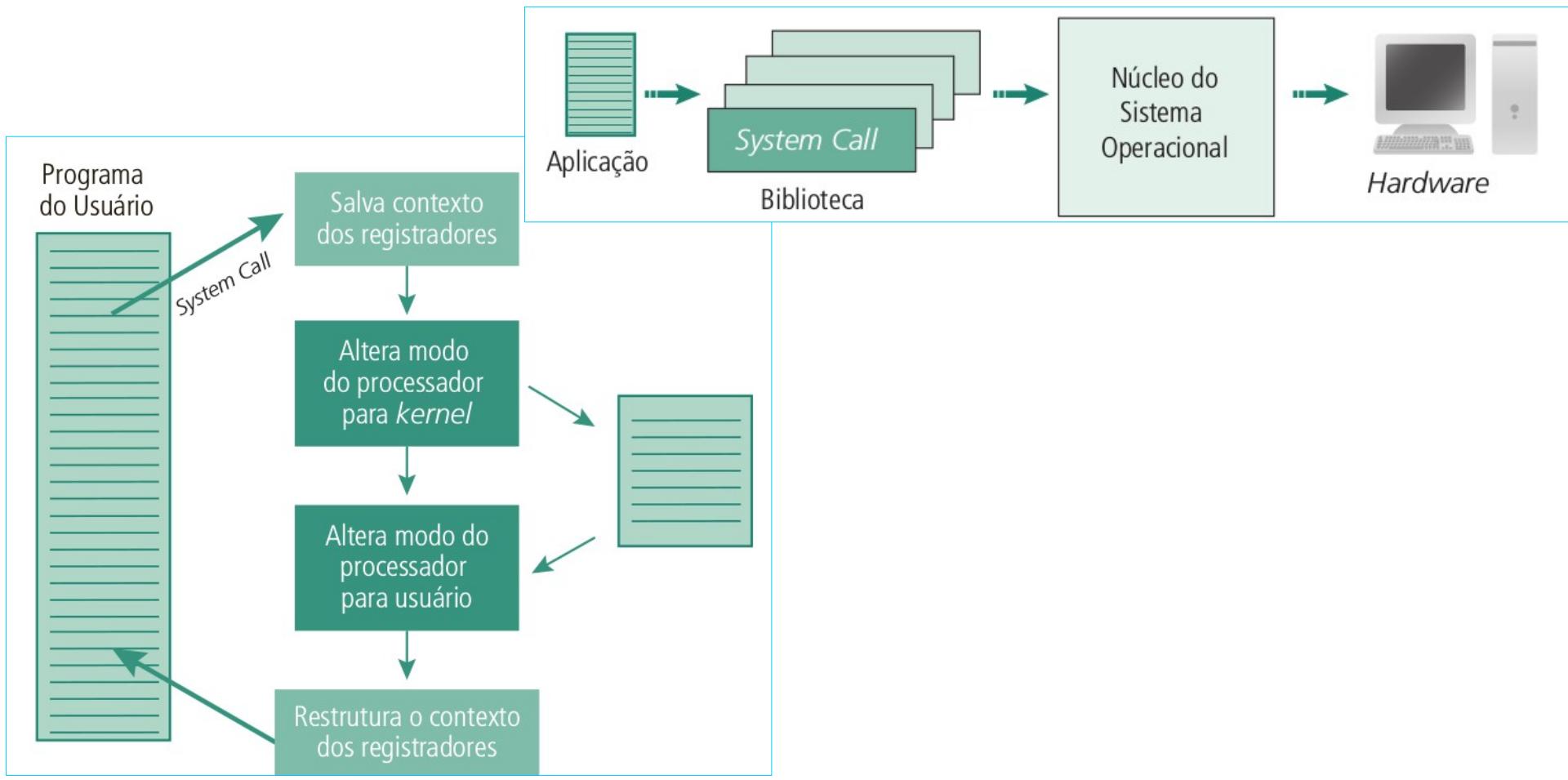
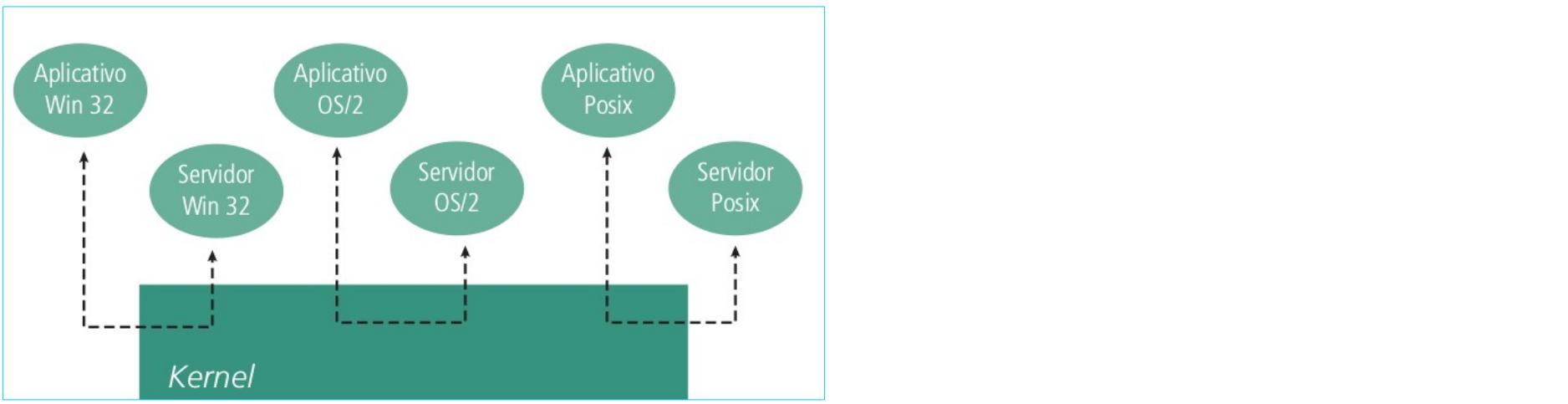
- O SO deve criar/alocar um para cada vez que um processo for criado
  - Registrador PC
  - Demais registradores
  - Variáveis



# - Valores que acompanham processos

- Chavear/alternar a CPU entre dois processos pode ser igualmente caro





# Criação de processos

- Principais eventos que levam à criação de processos
  - 1) Início do sistema operacional
  - 2) Execução de chamada ao sistema de criação de processos por um processo em execução
  - 3) Solicitação do usuário para criar um novo processo
  - 4) Início de um job em lote

# Exemplos simples de comunicação entre processos

Bianca Portes

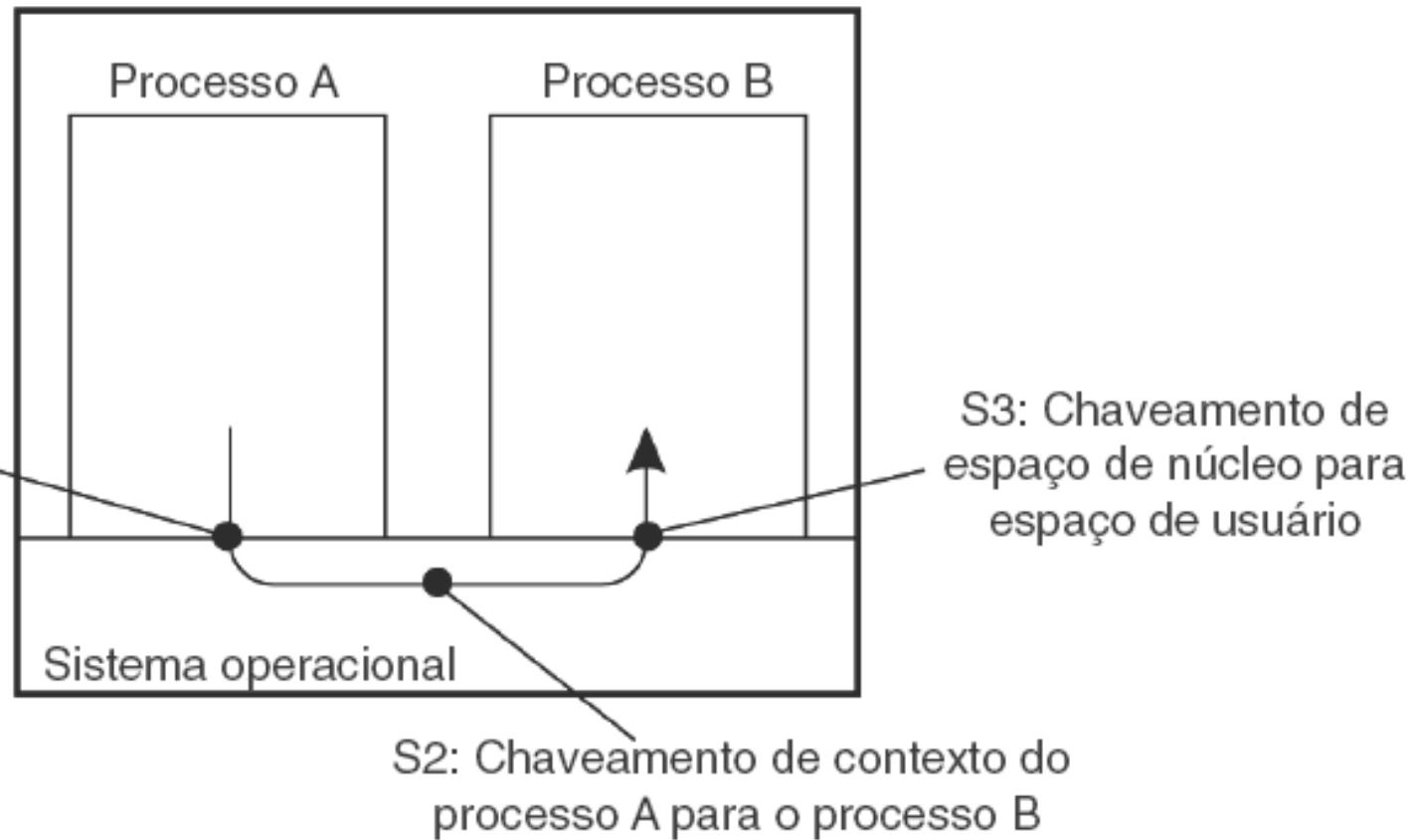
[bianca.castro@ifsudestemg.edu.br](mailto:bianca.castro@ifsudestemg.edu.br)

# Comunicação entre processos

- Há várias formas de comunicar processos
  - Sinais
  - Fluxos padrão
  - Socket
  - ...

# Comunicação entre processos

- Há várias formas de comunicar processos
  - Sinais
  - Fluxos padrão
  - Socket
  - ...



# 1- Comunicação por sinal

- Sinais são uma forma de comunicação entre processos em si e entre processos e o sistema operacional
- Existe um conjunto de sinais fixos que podem ser enviados
  - Eles permitem que um processo envie uma notificação para outro processo ou para o sistema operacional, indicando uma condição específica

## Standard signals

First the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from <a href="#">abort(3)</a>
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGNALRM	14	Term	Timer signal from <a href="#">alarm(2)</a>
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

Resumindo: quando um sinal é enviado a um processo, ele toma uma determinada ação dependendo do valor que este sinal carrega

# 1- Comunicação por sinal

- Exemplo de sinal
  - SIGINT: é enviado para um processo quando o usuário pressiona as teclas CTRL+C no terminal
    - Esse sinal geralmente é usado para solicitar que o processo termine sua execução de forma limpa

## Standard signals

First the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
<b>SIGHUP</b>	1	Term	Hangup detected on controlling terminal or death of controlling process
<b>SIGINT</b>	2	Term	Interrupt from keyboard
<b>SIGQUIT</b>	3	Core	Quit from keyboard
<b>SIGILL</b>	4	Core	Illegal Instruction
<b>SIGABRT</b>	6	Core	Abort signal from <a href="#">abort(3)</a>
<b>SIGFPE</b>	8	Core	Floating point exception
<b>SIGKILL</b>	9	Term	Kill signal
<b>SIGSEGV</b>	11	Core	Invalid memory reference
<b>SIGPIPE</b>	13	Term	Broken pipe: write to pipe with no readers
<b>SIGNALRM</b>	14	Term	Timer signal from <a href="#">alarm(2)</a>
<b>SIGTERM</b>	15	Term	Termination signal
<b>SIGUSR1</b>	30,10,16	Term	User-defined signal 1
<b>SIGUSR2</b>	31,12,17	Term	User-defined signal 2
<b>SIGCHLD</b>	20,17,18	Ign	Child stopped or terminated
<b>SIGCONT</b>	19,18,25	Cont	Continue if stopped
<b>SIGSTOP</b>	17,19,23	Stop	Stop process
<b>SIGTSTP</b>	18,20,24	Stop	Stop typed at terminal
<b>SIGTTIN</b>	21,21,26	Stop	Terminal input for background process
<b>SIGTTOU</b>	22,22,27	Stop	Terminal output for background process

# 1- Comunicação por sinal

- É importante saber que um sinal é normalmente tratado de duas maneiras diferentes:
  - 1) O processo recebe um sinal e FAZ o seu tratamento executando alguma ação
  - 2) O processo recebe um sinal e o IGNORA

# Implementando o programa em C

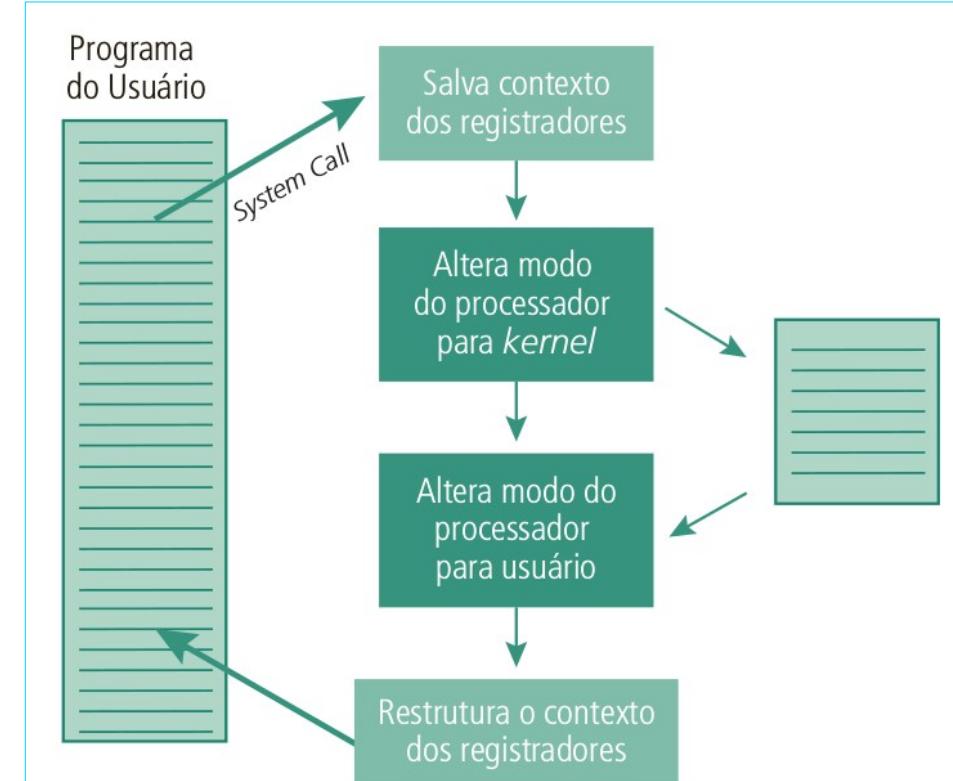
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <signal.h>
5
6 void trata_sinal(int sinal) {
7     printf("Sinal %d recebido. Encerrando...\n", sinal);
8     exit(0);
9 }
10
11 int main() {
12     // Registra o manipulador de sinal para SIGINT
13     signal(SIGINT, trata_sinal);
14
15     printf("Loop! Pressione CTRL+C para encerrar\n");
16     while(1) {
17         sleep(1);
18     }
19
20     return 0;
21 }
```

- A chamada de sistema *signal* é responsável pelo tratamento dos sinais recebidos por um processo. Esta função recebe dois parâmetros: o número do sinal *signum* e um manipulador *handler*

```
#include <signal.h>  
  
typedef void (*sighandler_t) (int)  
  
sighandler_t signal(int signum, sighandler_t handler);
```

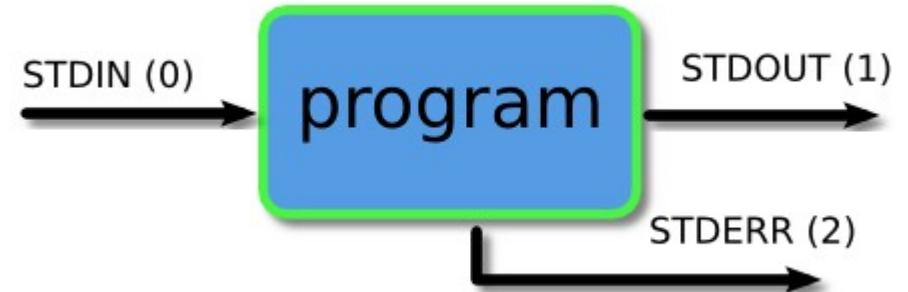
# Resultado?

- Qual foi o comportamento do processo quando você pressionou CTRL+C?
- O que acontece se você remover o ‘exit’ da função trata\_sinal? E se colocar ‘return’ no lugar? Por quê?
- Explique a ação que ocorreu utilizando a imagem ao lado (imagem apresentada na aula passada)
- Os sinais SIGKILL(9) e SIGSTOP(19) não podem ser tratados ou ignorados por esta função. Por quê?



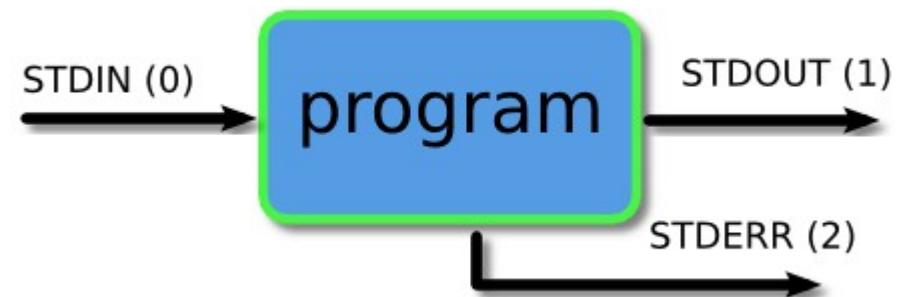
## 2- Fluxo padrão

- Sempre que precisamos capturar alguma informação ou exibir o resultado de um processo qualquer, o SO utiliza os fluxos padrão:
  - *stdin* - entrada padrão (usualmente o teclado)
  - *stdout* - saída padrão (usualmente o terminal de texto do monitor)
  - *stderr* - saída de erro (usualmente também o terminal de texto do monitor)
- Contudo, estas conexões podem ser "redirecionadas" para arquivos (> e >>) e de um programa para outro ( | )



## 2- Fluxo padrão

- Vamos fazer alguns testes no terminal?



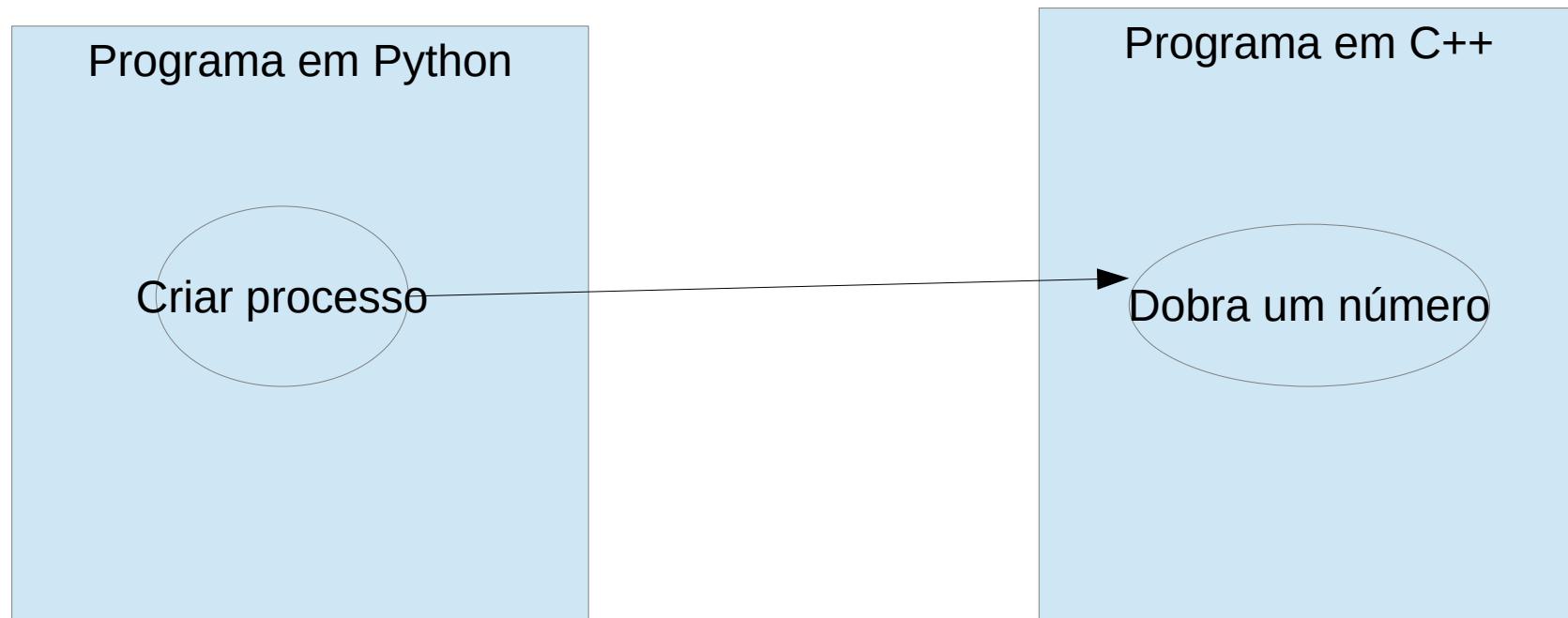
<https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/dev.html>

# Quem vai se comunicar?

- Vamos comunicar na própria máquina dois processos diferentes
  - Um processo em execução executará a chamada ao sistema de criação de processos
  - Esse processo criado (subprocesso) executará um código simples (dobrar um número) e retornará a resposta pelo canal de saída padrão, *stdout*

Para ser um pouco diferente, faremos um processo em Python criar um processo em C

# Fluxo padrão – exemplo 1



# Implementando do programa em C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int num;
7     scanf("%i", &num);
8     printf("Programa C: mensagem padrao - dobro de %i eh %i", num, 2*num);
9     perror("Programa C: mensagem na saida de erro! ");
10    sleep(10);
11    return 0;
12 }
```

Execute seu programa C e gere um executável com o nome “dobro.exe” na pasta ‘/semana2’

# Implementação em Python

- Em Python, é necessário importar a biblioteca de sistema *subprocess*
  - Provê uma forma genérica de criação e gerenciamento de processos na função Popen()
  - Ela permite a comunicação com o processo através de *pipes* (tubos/vias) do sistema operacional

```
import subprocess
```

# Implementação do programa em Python

```
1 import subprocess
2
3 args = ['dobro.exe']
4 p = subprocess.Popen(args,
5                      stdin=subprocess.PIPE,
6                      stdout=subprocess.PIPE,
7                      stderr=subprocess.PIPE,
8                      text=True)
9
10 #Envio de mensagem para alimentar o subprocesso
11 saida = p.communicate('8')
12
13 print("Saídas do subprocesso -")
14 print("\n\tSaida padrão: " + saida[0] + "\n\tSaida de erro: " + saida[1])
15
16 print("\nPrograma Python finalizado!")
```

Salve como '/semana2/chamadobrar.py'

# Implementação do programa em Python

```
1 import subprocess  
2  
3 args = ['dobra.exe']  
4 p = subprocess.Popen(args,  
5                      stdin=subprocess.PIPE,  
6                      stdout=subprocess.PIPE,  
7                      stderr=subprocess.PIPE,  
8                      text=True)  
9  
10 #Envio de mensagem para o processo  
11 saída = p.communicate()  
12  
13 print("Saídas do subprocesso")  
14 print("\n\tSaida padrão")  
15  
16 print("\nPrograma Python")  
  
class subprocess.Popen(args, bufsize=-1,  
executable=None, stdin=None, stdout=None,  
stderr=None, preexec_fn=None, close_fds=True,  
shell=False, cwd=None, env=None,  
universal_newlines=None, startupinfo=None,  
creationflags=0, restore_signals=True,  
start_new_session=False, pass_fds=(), *,  
encoding=None, errors=None, text=None)
```

Salve como '/semana2/chamadobrar.py'

# Resultado para ‘chamadobrar.py’

```
$ python chamadobrar.py
Saídas do subprocesso -
    Saída padrão: Programa C: mensagem padrao - dobro de 8 eh 16
    Saída de erro: Programa C: mensagem na saida de erro! : No error

Programa Python finalizado!
```

# Resultado?

- Qual a ordem da impressão dos textos?
- Qual foi o comportamento dos processos?
- Os processos são concorrentes?

# Criando processos

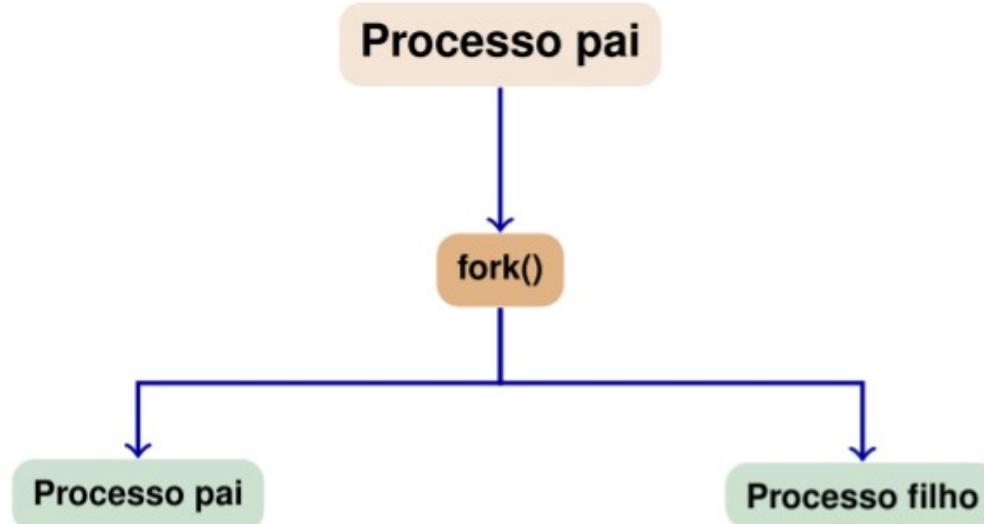
- No exemplo anterior, usamos subprocess
- Também podemos usar chamada de sistema de forma direta
  - Unix e Unix-like
    - Fork
  - Windows
    - CreateProcess

# Criando processos

- No exemplo anterior, usamos subprocess
- Também podemos usar chamada de sistema de forma direta
  - Unix e Unix-like
    - Fork
  - Windows
    - CreateProcess

# Fork

- É uma função do SO que cria uma cópia exata do processo atual (processo pai) e retorna o ID do processo criado (processo filho)
- Os dois processos são completamente independentes, com PIDs diferentes, mas têm exatamente o mesmo código
  - Após a chamada ao fork(), o processo filho e o processo pai executam a partir do ponto em que o fork() foi chamado



# Fluxo padrão – exemplo 2

```
1 import os
2
3 print("processo pai iniciando")
4 iPid = os.fork()
5 if iPid < 0: #Erro
6     print("Erro ao gerar o processo filho")
7 elif iPid == 0: #Processo filho
8     print("Sou o processo filho gerado: ", iPid)
9 else: #Processo pai
10    print("Sou o processo pai: ", iPid)
```

# Resultado?

- Qual a ordem da impressão dos textos?
- Qual foi o comportamento dos processos?
- Os processos são concorrentes?

Pesquise como usar a comunicação via PIPE entre os processos pai e filho

# Vamos praticar!!

É possível ver o estado do sistema Linux usando os comandos *top* ou *uptime*

```
biapc@biapc-U24:~$ uptime  
10:06:17 up 34 min, 1 user, load average: 0,15, 0,20, 0,28
```

Hora atual; tempo do sistema ativo; nº de usuários logados; nº médio de processos prontos para ser executados nos últimos 1, 5 e 15 minutos

1) Utilizando a biblioteca subprocess no Python, faça um programa que:

a) Diga se taxa de ocupação do sistema está ok (carga média  $\leq 1$ ) ou se há muita concorrência (carga média  $> 1$ )

Se precisar de mais ajuda com Python:

- Sintaxe básica do Python: <http://devfuria.com.br/python/sintaxe-basica/>
- Documentação Python: <https://docs.python.org/pt-br/3/reference/>

# Vamos praticar!!

2) Gere um programa que, após ser alimentado com o alvo da busca, abra o firefox e ensine o usuário a buscar no google com o  
<http://letmegooglet.com>

3) Faça um programa que verifique se as dependências do seu programa estão instaladas, se não, instale-as.

Verificar pacote: pip show <pacote>

Instalar pacote: pip install <pacote>

# Compartilhamento de memória - introdução a Threads

Bianca Portes

[bianca.castro@ifsudestemg.edu.br](mailto:bianca.castro@ifsudestemg.edu.br)

# Recordando...

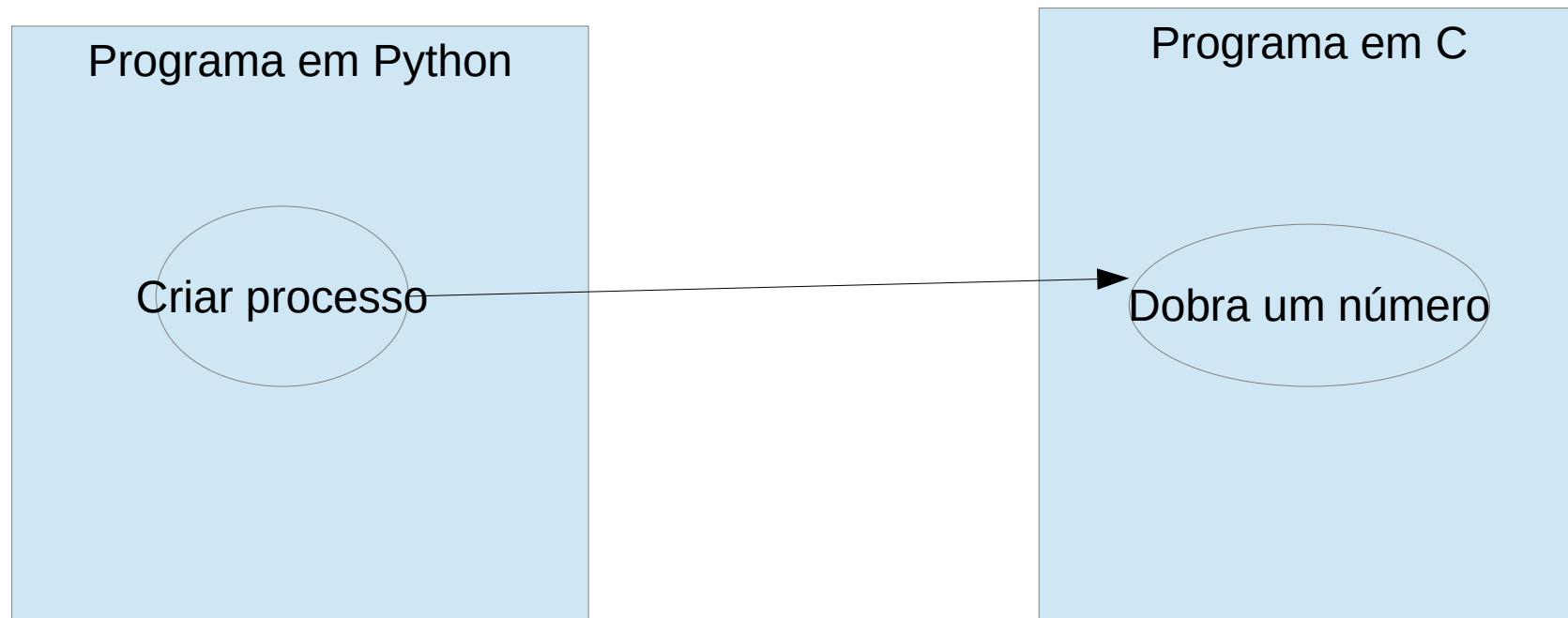
- Pretendemos responder as seguintes perguntas

- O que é um processo?
  - Qual a importância dos processos?

Aula de  
processos

Vimos formas de **comunicação entre processos** e utilizamos algumas delas, a por **fluxo padrão** e **signal**

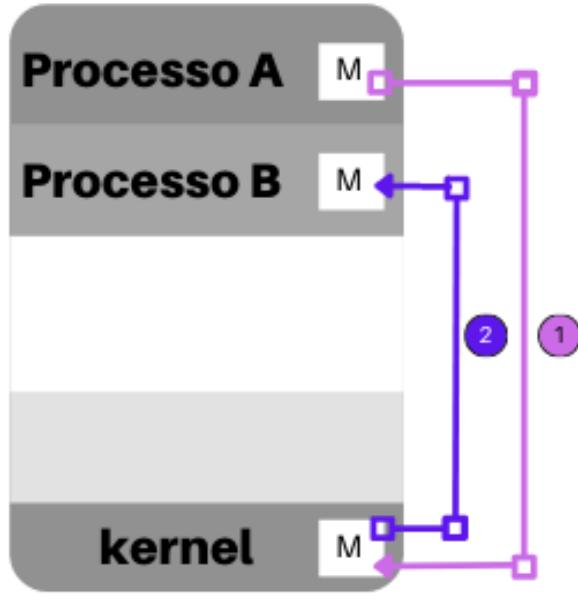
# Recordando...



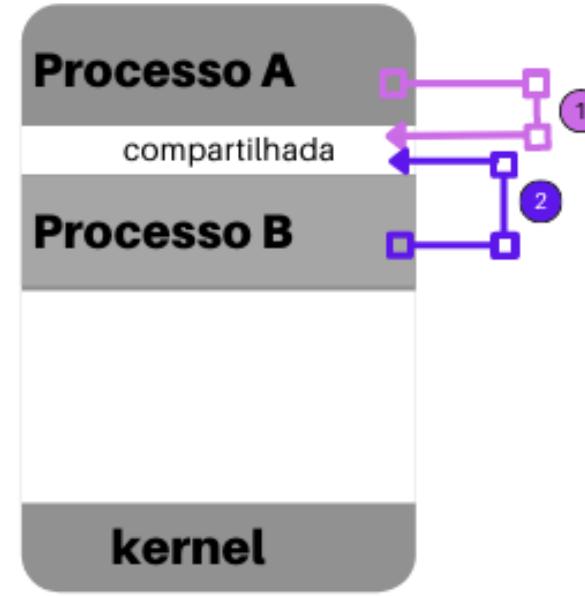
Vários mecanismos de **comunicação entre processos** estão disponíveis para sistemas POSIX, inclusive a memória compartilhada

Vários mecanismos de **comunicação entre processos** estão disponíveis para sistemas POSIX, inclusive a **memória compartilhada**

Vimos em Sistemas Operacionais! Está lembrado?



(a)



(b)

- Troca de pequenas quantidades de dados
- Não é necessário evitar conflitos
- Fácil de implementar
- Velocidade máxima de comunicação

# Exemplo ilustrativo de uso da API POSIX para memória compartilhada com a linguagem C

```
1 #include <stdio.h>
2 #include <sys/shm.h>
3 #include <sys/stat.h>
4
5 int main(){
6     /* o identificador do segmento de memória compartilhada */
7     int segment_id;
8     /* um ponteiro para o segmento de memória compartilhada */
9     char *shared_memory;
10
11    /* o tamanho (em bytes) do segmento de memória compartilhada */
12    const int size = 4096;
13    /* aloca um segmento de memória compartilhada */
14    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
15
16    /* por cmd, redirecione por pipe esse id pro processo desejado */
17    printf("%i", segment_id);
18
19    /* anexa o segmento de memória compartilhada */
20    shared_memory = (char *) shmat(segment_id, NULL, 0);
21    /* grava uma mensagem no segmento de memória compartilhada */
22    sprintf(shared_memory, "Oi, pessoal!");
23
24    return 0;
25 }
```

**Processo 2:** aloca, lê e imprime o que está armazenado na memória!

**Processo 1:** aloca e escreve na memória!

```
1 #include <stdio.h>
2 #include <sys/shm.h>
3
4
5 int main(){
6     /* o identificador do segmento de memória compartilhada */
7     int segment_id;
8     /* um ponteiro para o segmento de memória compartilhada */
9     char *shared_memory;
10
11    scanf("%i", &segment_id);
12
13    /* anexa o segmento de memória compartilhada */
14    shared_memory = (char *) shmat(segment_id, NULL, 0);
15
16    /* exibe a cadeia de caracteres a partir da memória compartilhada */
17    printf("%s\n", shared_memory);
18
19    /* desanexa o segmento de memória compartilhada */
20    shmdt(shared_memory);
21    /* remove o segmento de memória compartilhada */
22    shmctl(segment_id, IPC_RMID, NULL);
23 }
```

[1] <https://man7.org/linux/man-pages/man2/shmget.2.html>

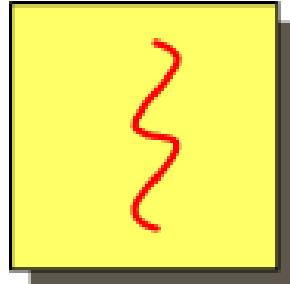
[2] Fundamentos de Sistemas Operacionais (Silberschatz), pág. 76

Nossa conversa sobre formas de comunicação entre processos ainda não acabou!

Contudo, aproveitando que falamos de memória compartilhada, que tal falarmos de *Threads*?

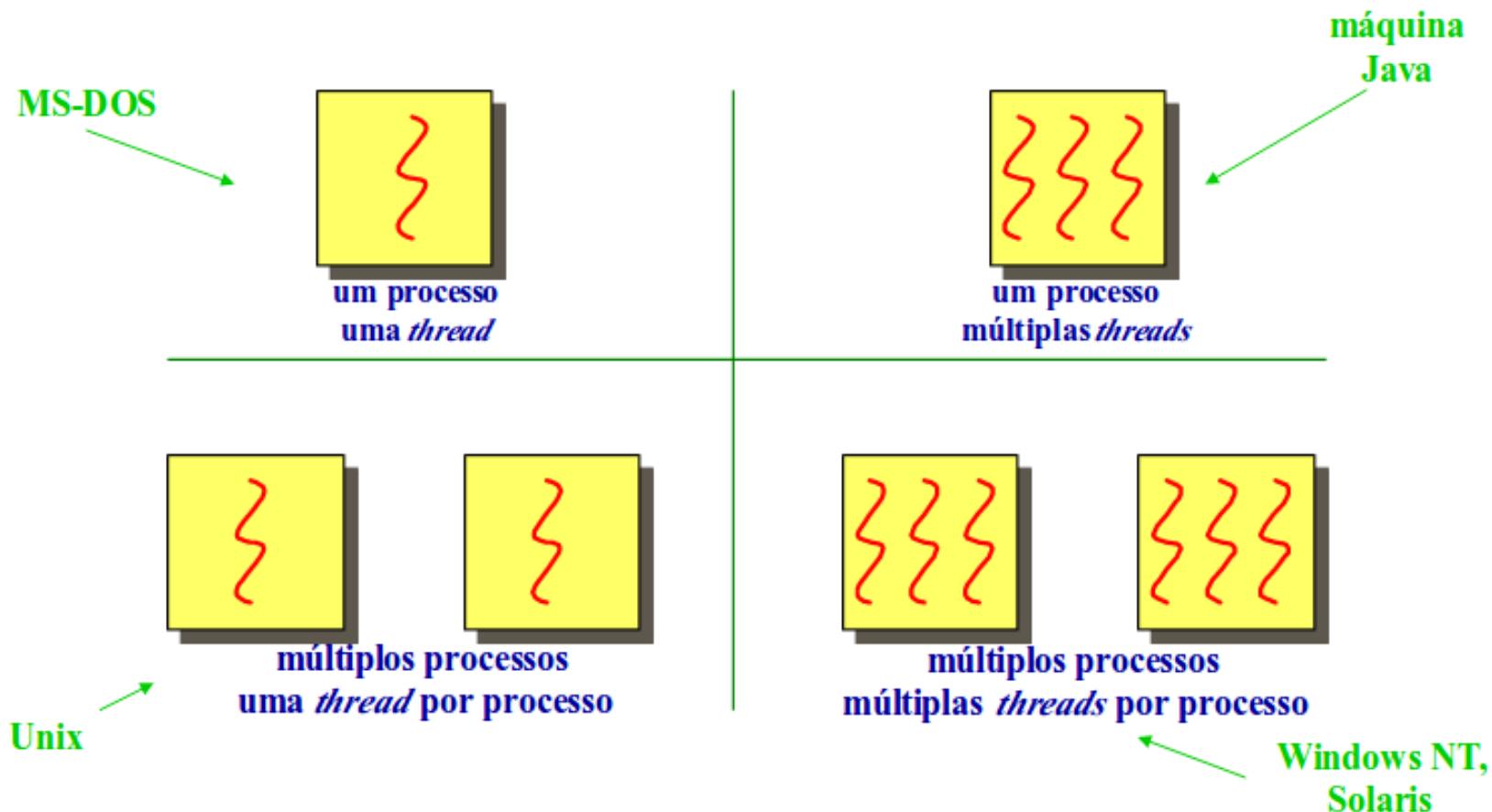
# Threads

- Até o momento, vimos um processo como um programa em execução com um único *thread* de controle



um processo  
uma *thread*

- Um *thread* é uma unidade básica de utilização da CPU
- Sistemas operacionais modernos fornecem recursos que permitem que um processo contenha várias *threads*



- Quando múltiplos *threads* coexistem em um processo, é possível:
  - dividir entre eles a memória alocada para este processo
  - executar mais de uma tarefa ao mesmo tempo

# Thread

- Um *thread* é uma unidade básica de utilização da CPU
  - Composto
    - Identificador do *thread*
    - Registrador PC
    - Demais registradores
    - Pilha de memória
  - Compartilha com outros *threads* irmãos
    - Seção de código e dados
    - Recursos do SO (ex.: arquivos abertos, sinais)

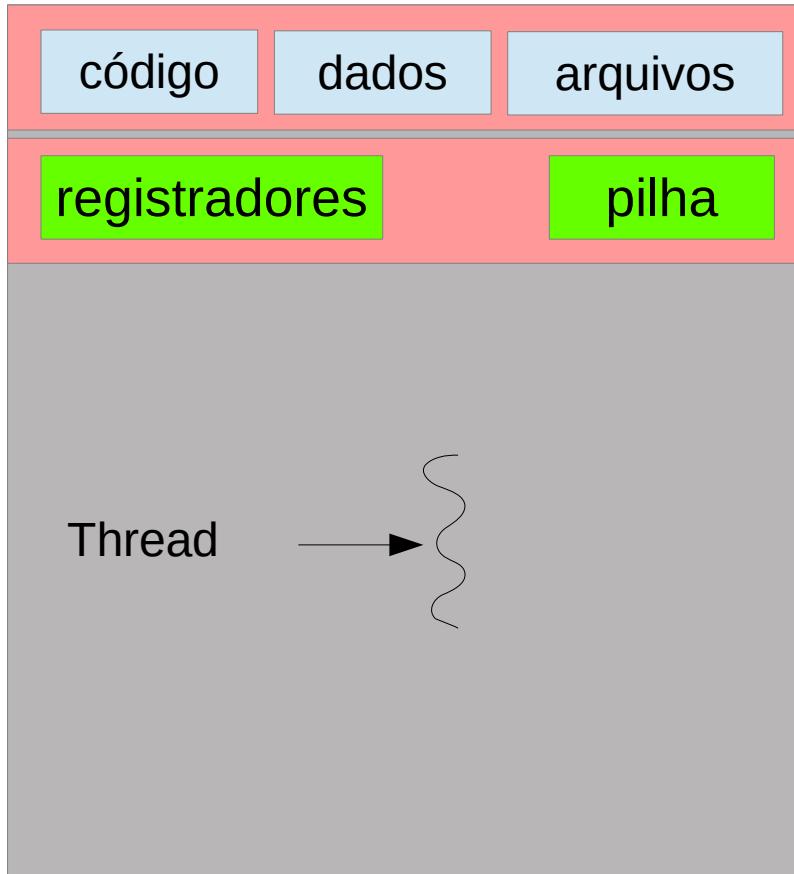
# Thread

- Um *thread* é uma unidade básica de utilização da CPU
  - Composto
    - Identificador do *thread*
    - Registrador PC
    - Demais registradores
    - Pilha de memória
  - Compartilha com outros *threads* irmãos
    - Seção de código e dados
    - Recursos do SO (ex.: arquivos abertos, sinais)

***Threads* seriam processos leves no sentido que:**

- oferecem vantagens similares a processos;
- não requerem típicas técnicas de comunicação de processos

Processo com um único *thread*



Processo com vários *threads*



Observe que fica a cargo da aplicação gerenciar o compartilhamento de dados e recursos com vários *threads*

## - Threads

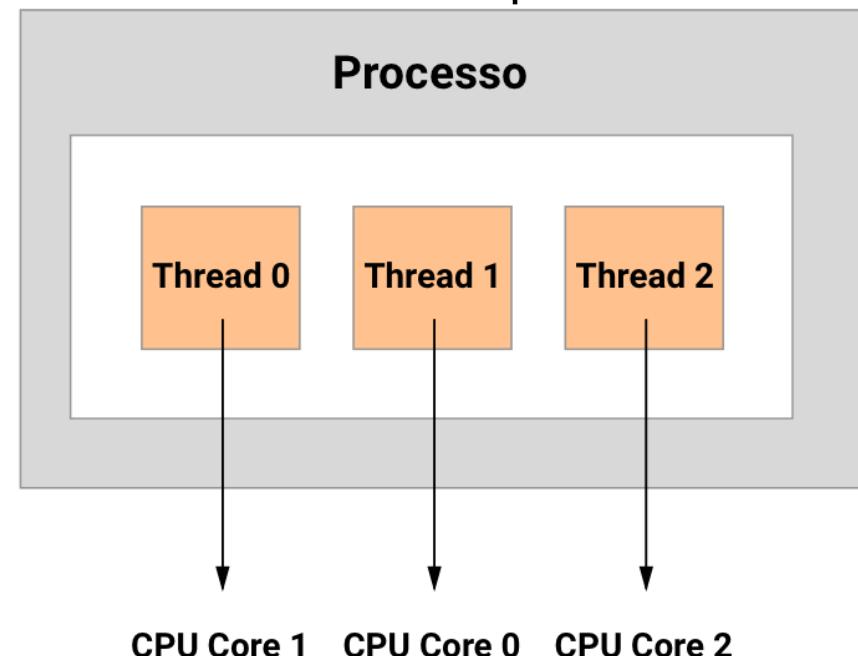
Como em outros modelos de memória compartilhada, em *threads*:

- eles não são automaticamente protegidos uns contra os outros, como acontece com os processos por troca de mensagens
- o desenvolvimento de aplicações *multithread* requer esforço intelectual adicional
- boas práticas: elaborar adequadamente o projeto e manter as coisas simples

**Resumindo:** nenhuma tentativa é feita para conseguir alto grau de transparência de concorrência se isso resultar em degradação do desempenho

# Motivação de uso de threads

- Aumento de capacidade de resposta
  - Se for executada uma chamada bloqueadora, o processo não é bloqueado como um todo
- Economia na alocação de memória e recursos - é mais econômica a troca de contexto
  - No Solaris,
    - a criação de um processo é cerca de 30x mais lenta do que a criação de um *thread*
    - a mudança de contexto entre *threads* é cerca de 5x mais rápida
- Compartilhamento de recursos
- Escalabilidade
  - Aumento do paralelismo



# Com usar? A forma mais simples...

A forma mais simples de usar *thread* é instanciá-la em uma função. Precisou? Basta chamar o método para que o ele inicie o trabalho (*job*)

Espera-se com o uso de *threads* que as tarefas sejam executadas de forma concomitante

# Implementação



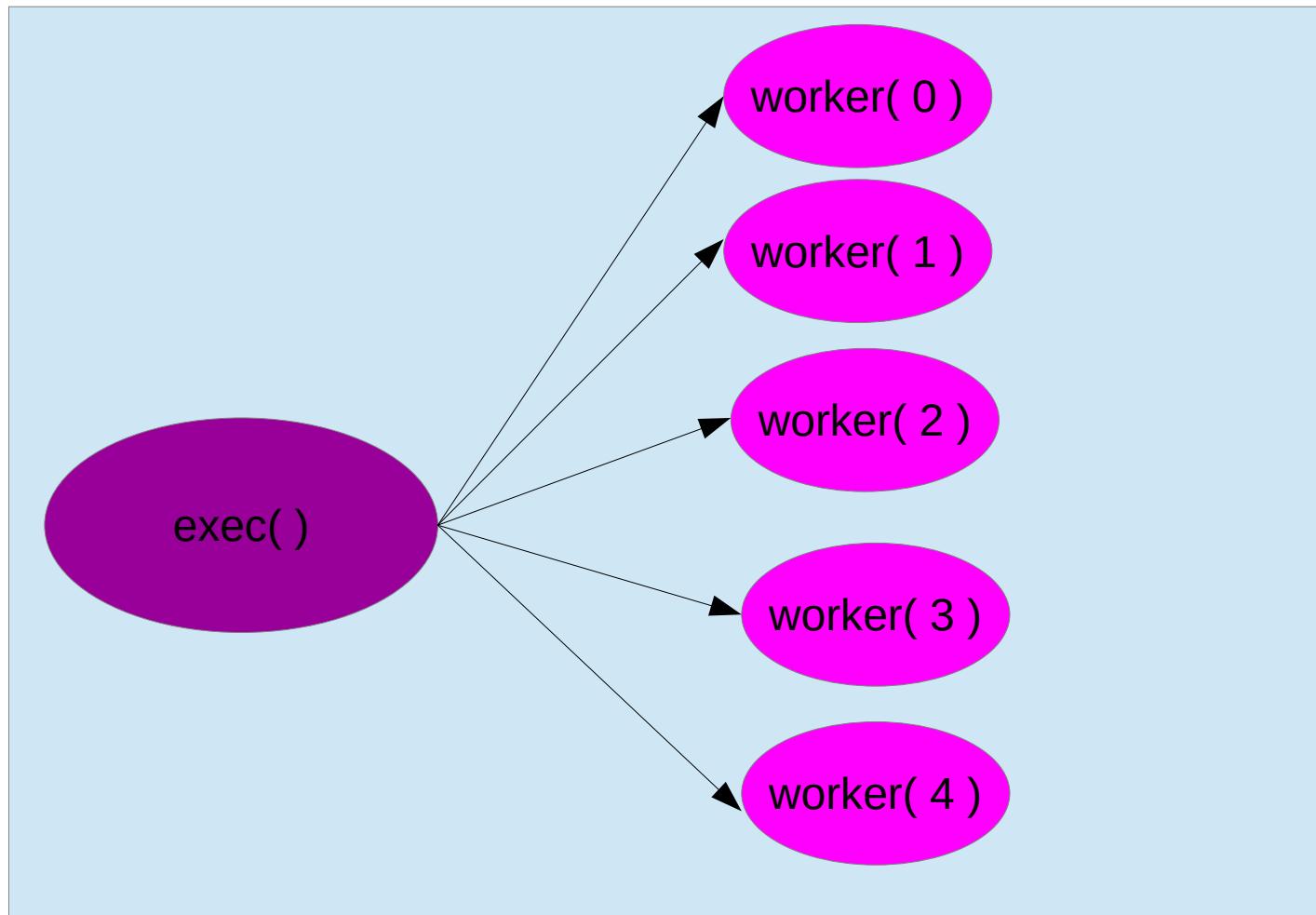
módulo nativo para criar *threads*

*threading*

Para fazer a importação do módulo, adicione ao seu script:

```
import threading
```

# Implementação



# Implementação...

módulos  
importados

```
import threading  
import time
```

Além do módulo *threading*, importaremos o módulo *time*

worker

```
def worker(thread_num):  
    print("Função chamada pela thread nº %i começando\n" % thread_num)  
  
    #Apenas para simular uma carga de trabalho que demora 1 segundo  
    time.sleep(5)  
    print("Aguardando a thread %i acabar...\n" % thread_num)
```

Aqui criamos um método simples, o método *worker*

Para simular carga de trabalho, faremos ele aguardar 5 segundos antes de terminar → instrução *time.sleep(5)*

# Implementação...

A *thread worker* não vai executar até ser iniciada!

```
t = threading.Thread(target=worker, args=(i,))
t.start()
```

Parâmetros da classe *Thread*:

- *target*:
  - É o alvo de trabalho da *thread*!
  - Recebe o nome do método que deve ser executado quando a *thread* tiver sua atividade iniciada
- *args*:
  - Recebe a lista de argumentos que serão passados para o alvo de trabalho da *thread*

Pra organizar, vamos fazer o método **exec** como o responsável por criar as 5 *threads work*

```
import threading
import time

def worker(thread_num):
    print("Função chamada pela thread nº %i começando\n" % thread_num)

    #Apenas para simular uma carga de trabalho que demora 1 segundo
    time.sleep(5)
    print("Aguardando a thread %i acabar...\n" % thread_num)

def exec():
    for i in range(5):
        t = threading.Thread(target=worker, args=(i,))
        t.start()
    print("Acabou!!!")

if __name__ == "__main__":
    exec()
```

# Exemplo de resultado de execução...

Função chamada pela thread nº 0 começando

Função chamada pela thread nº 3 começando

Função chamada pela thread nº 1 começando

Função chamada pela thread nº 4 começando

Função chamada pela thread nº 2 começando

**Acabou!!!**

Aguardando a thread 0 acabar...

Aguardando a thread 3 acabar...

Aguardando a thread 2 acabar...

Aguardando a thread 4 acabar...

Aguardando a thread 1 acabar...

# Questões!

1) Quais foram as ordens de impressão na sua máquina?

Repita a operação e verifique se a ordem continua a mesma.

2) Por que o programa se comporta assim?

# Sincronizando um pouco...

```
import threading
import time

def worker(thread_num):
    print("Função chamada pela thread nº %i começando\n" % thread_num)

    #Apenas para simular uma carga de trabalho que demora 1 segundo
    time.sleep(5)
    print("Aguardando a thread %i acabar...\n" % thread_num)

def exec():
    threads = [] #Lista para armazenar as 5 threads que criaremos
    for i in range(5):
        t = threading.Thread(target=worker, args=(i,))
        threads.append(t) #Armazenando a thread criada na lista
        t.start()

    for i in threads:
        i.join()

    print("Acabou!!!")

if __name__ == "__main__":
    exec()
```

# Sincronizando um pouco...

```
import threading
import time

def worker(thread_num):
    print("Função chamada pela thread nº %i começando\n" % thread_num)

    #Apenas para simular uma carga de trabalho que demora 1 segundo
    time.sleep(5)
    print("Aguardando a thread %i acabar...\n" % thread_num)

def exec():
    threads = [] #Lista para armazenar as 5 threads que criaremos
    for i in range(5):
        t = threading.Thread(target=worker, args=(i,))
        threads.append(t) #Armazenando a thread criada na lista
        t.start()

    for i in threads:
        i.join()

    print("Acabou!!!")

if __name__ == "__main__":
    exec()
```

exec

```
def exec():
    threads = [] #Lista para armazenar as 5 threads que criaremos
    for i in range(5):
        t = threading.Thread(target=worker, args=(i,))
        threads.append(t) #Armazenando a thread criada na lista
        t.start()

    for i in threads:
        i.join()

    print("Acabou!!!")
```

Aqui criamos o método **exec**

1º *for* → inicia 5 *threads* que executarão, cada uma,o método **worker**

2º *for* → aguardar que todas as 5 *threads* finalizem o seu trabalho

# Exemplo de resultado de execução...

Função chamada pela thread nº 0 começando

Função chamada pela thread nº 3 começando

Função chamada pela thread nº 1 começando

Função chamada pela thread nº 4 começando

Função chamada pela thread nº 2 começando

Aguardando a thread 0 acabar...

Aguardando a thread 3 acabar...

Aguardando a thread 2 acabar...

Aguardando a thread 4 acabar...

Aguardando a thread 1 acabar...

**Acabou!!!**

# Questões!

- 1) O que mudou em relação às impressões na tela?
- 2) Por que houve a mudança?
- 3) O que o método *join* faz?

Poste suas respostas  
na nossa sala! =)



**Q1(ENADE)** - Uma alternativa para o aumento de desempenho de sistemas computacionais é o uso de processadores com múltiplos núcleos, chamados multicores. Nesses sistemas, cada núcleo, normalmente, tem as funcionalidades completas de um processador, já sendo comuns, atualmente, configurações com 2, 4 ou mais núcleos.

Com relação ao uso de processadores multicores, e sabendo que threads são estruturas de execução associadas a um processo, que compartilham suas áreas de código e dados, mas mantêm contextos independentes, analise as seguintes asserções.

I) Ao dividirem suas atividades em múltiplas threads que podem ser executadas paralelamente, aplicações podem se beneficiar mais efetivamente dos diversos núcleos dos processadores multicores

porque

II) o sistema operacional nos processadores multicores pode alocar os núcleos existentes para executar simultaneamente diversas seqüências de código, sobrepondo suas execuções e, normalmente, reduzindo o tempo de resposta das aplicações às quais estão associadas.

Acerca dessas asserções, assinale a opção correta.

A ) As duas asserções são proposições verdadeiras, e a segunda é uma justificativa correta da primeira.

B) As duas asserções são proposições verdadeiras, mas a segunda não é uma justificativa correta da primeira.

C) A primeira asserção é uma proposição verdadeira, e a segunda, uma proposição falsa.

D) A primeira asserção é uma proposição falsa, e a segunda, uma proposição verdadeira.

E) Tanto a primeira quanto a segunda asserções são proposições falsas.

**Q2 (ENADE)** - Uma antiga empresa de desenvolvimento de software resolveu atualizar toda sua infraestrutura computacional adquirindo um sistema operacional multitarefa, processadores multi-core (múltiplos núcleos) e o uso de uma linguagem de programação com suporte a threads.

O sistema operacional multitarefa de um computador é capaz de executar vários processos (programas) em paralelo. Considerando esses processos implementados com mais de uma thread (multi-threads), analise as afirmações abaixo.

- I. Os ciclos de vida de processos e threads são idênticos.
- II. Threads de diferentes processos compartilham memória.
- III. Somente processadores multi-core são capazes de executar programas multi-threads.
- IV. Em sistemas operacionais multitarefa, threads podem migrar de um processo para outro.

É correto apenas o que se afirma em

- A) I.
- B) II.
- C) I e III.
- D) I e IV.
- E) II e IV.

# *Speedup*

Bianca Portes

[bianca.castro@ifsudestemg.edu.br](mailto:bianca.castro@ifsudestemg.edu.br)

# Recordando...

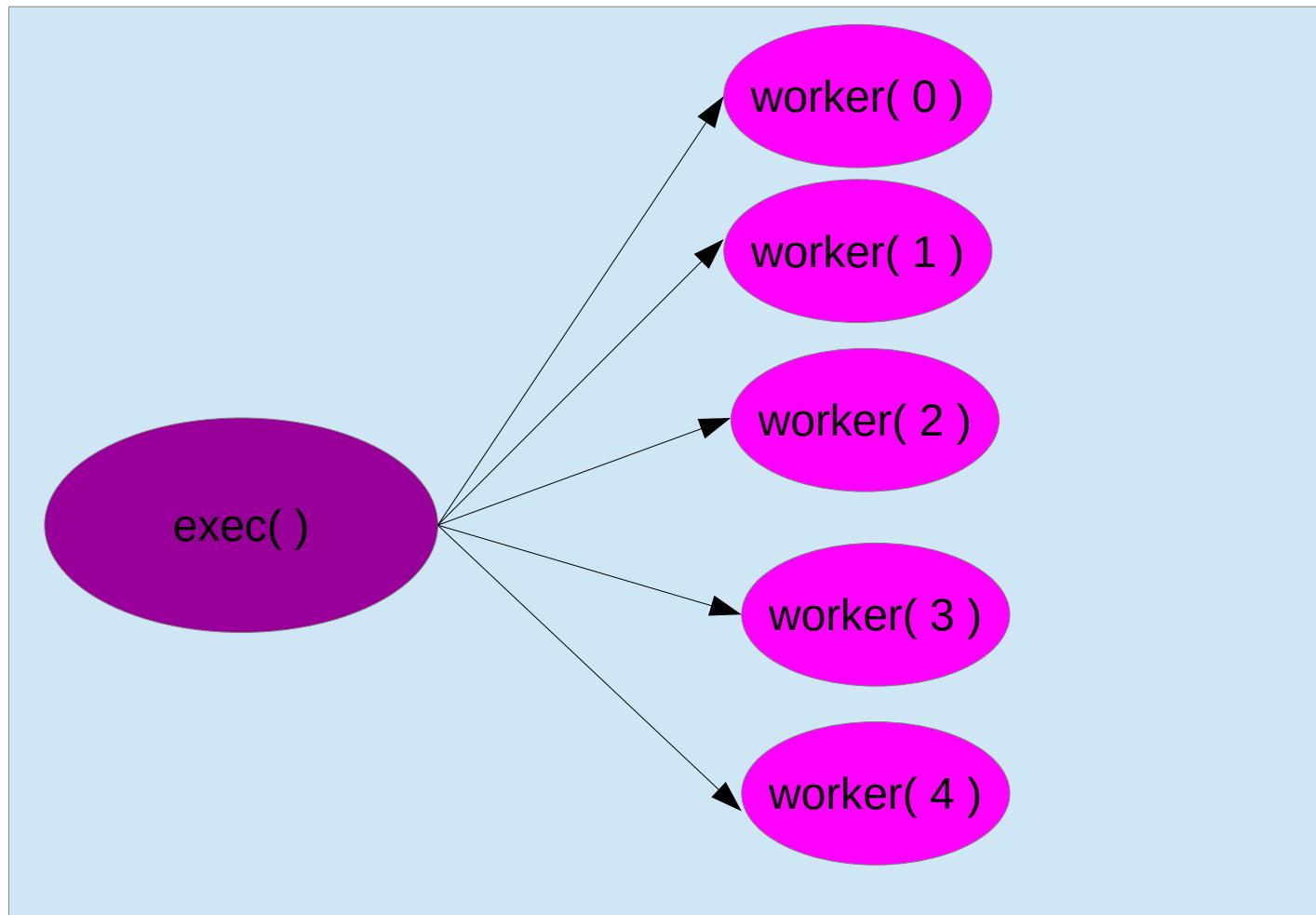
- Pretendemos responder as seguintes perguntas

- O que é um thread?
  - Qual a diferença entre processos e threads

Aula de  
threads

Vimos como criar threads simples e até a gerenciar com pontos de sincronismo!

# Recordando...



# Aula de hoje

- A aula de hoje objetiva
  - fornecer métricas comparativas entre algoritmos
  - introduzir o pensamento de adição de paralelismo

# Concorrência x Parallelismo

Podem parecer a mesma coisa, mas não são!!

- Parallelismo ocorre quando duas ou mais tarefas são executadas literalmente ao mesmo tempo
  - Exemplo: tarefas executadas em núcleos de processamento distintos
- Concorrência ocorre quando duas ou mais tarefas têm o tempo sobreposto, mas não significando que estejam executando necessariamente no mesmo instante
  - Exemplo: pipeline e escalonador

*“Concurrency is about **dealing** with lots of things at once. Parallelism is about **doing** lots of things at once.”* (Rob Pike)

ou

“Concorrência é sobre **lidar** com várias coisas ao mesmo tempo.  
Paralelismo é sobre **fazer** várias coisas ao mesmo tempo.”

# Concorrência x Parallelismo

- Na prática, não temos controle para saber se um paralelismo real está ocorrendo dentro de uma mesma máquina
- Como saber se houve algum ganho de desempenho em relação ao sequêncial?

Para isso, é preciso estabelecer um **critério ou métrica para comparar !**

# Métricas de desempenho

- **Tempo de resposta ou execução**
  - Quanto tempo demora para executar uma tarefa?
- **Vazão (throughput)**
  - Quanto trabalho foi feito por unidade de tempo?  
Ex.: tarefas ou transações por hora

# Métricas de desempenho

- **Tempo de resposta ou execução**
  - Quanto tempo demora para executar uma tarefa?
- **Vazão (throughput)**
  - Quanto trabalho foi feito por unidade de tempo?  
Ex.: tarefas ou transações por hora

Você já deve ter usado algum benchmark para obter dado bruto sobre algo que testou, correto?

As questões que importam para esse resultado obtido:

- ele sozinho faz algum sentido?
- como interpretá-lo?

# Métricas de desempenho

- **Tempo de resposta ou execução**
  - Quanto tempo demora para executar uma tarefa?
- **Vazão (throughput)**
  - Quanto trabalho foi feito por unidade de tempo?  
Ex.: tarefas ou transações por hora

Tomando o **tempo de execução** como o dado bruto obtido para a execução de **dois algoritmos voltados para a mesma tarefa**. Como você interpreta esses dados?

# Comparando algoritmos

- O objetivo do paralelismo é obter desempenho superior em relação à versão do algoritmo anterior

Tempo de execução = tempo de computação +  
tempo ocioso +  
tempo de comunicação

As medidas independentes não são triviais de serem obtidas...

Mas **tempo de execução** é trivial de ser obtido

# Comparando algoritmos

- ***Speedup*** é a razão do tempo de execução do algoritmo anterior  $T(n)$  com o tempo de execução do algoritmo paralelo  $T(n,p)$

$$S(n,p) = \frac{T(n)}{T(n,p)}$$

- $T(n)$ : tempo de execução do algoritmo anterior (serial, por exemplo)
- $T(n,p)$ : tempo de execução do algoritmo paralelo em  $p$  processadores da máquina paralela

# Comparando algoritmos

- ***Speedup*** é a razão do tempo de execução do algoritmo anterior  $T(n)$  com o tempo de execução do algoritmo paralelo  $T(n,p)$

$$S(n,p) = \frac{T(n)}{T(n,p)}$$

Exemplo: se um algoritmo sequencial leva 10 segundos, e o otimizado leva 2 segundos:

$$\text{Speedup} = 10/2 = 5$$

Ou seja, a versão otimizada é 5 vezes mais rápida.

# Comparando algoritmos

O *speedup* fornece um indicador para o aumento da velocidade por utilizar mais recursos da máquina paralela

$$0 < S(n, p) < p$$

- Se  $S(n,p) = p$ , teremos um *speedup* linear
- Speedup linear é raro, pois a maioria das soluções paralelas introduzem alguma sobrecarga que é produto da distribuição de carga e da comunicação entre os processos

# Comparando algoritmos

O *speedup* fornece um indicador para o aumento da velocidade por utilizar mais recursos da máquina paralela

$$0 < S(n, p) < p$$

- Se  $S(n,p) = p$ , teremos um *speedup* linear

Exemplo: se em uma máquina com 4 núcleos, um algoritmo sequencial leva 8000ms segundos e o paralelo leva 2000 ms:

$$\text{Speedup}(n,4)=8000/2000 = 4$$

Ou seja, sugere que a paralelização foi eficiente, atingindo o limite ideal dos 4 núcleos nesse caso

# Atenção!

- Tente garantir na medida do possível exclusividade na utilização do processador para conseguir medidas mais confiáveis



# Cálculo de tempo

- Para achar o tempo de execução do processo:
  - ache o tempo início ( $t_1$ ) e o tempo de fim ( $t_2$ ) para a execução do processo
- Por quê?
  - **Ambiente não-determinístico:** sistemas operacionais, garbage collector, cache, variações de CPU, etc
  - **Ruído externo:** notificações, processos em background, turbo boost do processador...
  - **Minimizar outliers:** repetir suaviza medições atípicas (ex: uma execução demorada por interferência)

# Cálculo de tempo

- A melhor forma de achar um tempo representativo de execução do processo:

Executá-lo um número fixo de vezes e utilizar a média dos tempos de execução para o speedup

- Um script ajuda muito nessa tarefa!

Estatisticamente falando, qual o valor para esse número de vezes?

# Cálculo de tempo

- A melhor forma de achar um tempo representativo de execução do processo:

Executá-lo um número fixo de vezes e utilizar a média dos tempos de execução para o speedup

- Um script ajuda muito nessa tarefa!

Estatisticamente falando, qual o valor para esse número de vezes?

A resposta é: depende! Quanto menor o tempo individual de execução, menor é a precisão numérica que o computador provê para aquele tempo fornecido. Assim, quanto menor o tempo, mais execuções serão necessárias

- Contudo, para uma precisão estatística com 95% de confiança, 30 amostras costumam ser suficientes

# Construindo um benchmark confiável

Se você deseja comparar dois hardwares distintos ou duas versões de algoritmos distintos, você deve fazer os passos abaixo para cada um deles:

- 1) Rode o algoritmo 30+ vezes
- 2) Descarte 5 % primeiras execuções (*warm-up*, aquecimento da JVM ou JIT)

Esse *warm-up* (ou "aquecimento") tende a causar tempos instáveis nas primeiras execuções

- 3) Calcule média para usá-la como dado bruto no Speedup

Seria ainda melhor se, para não ficar apenas na performance central, você calculasse também o desvio padrão e o intervalo de confiança

```
1 import threading
2 import time
3
4 def worker(thread_num):
5     print("Iniciou-se a execução do thread ", thread_num)
6     time.sleep(5)
7     print("Acabou a execução do thread ", thread_num)
8
9 def exec():
10    threads= []
11
12    for i in range(5):
13        t = threading.Thread(target=worker, args=(i,))
14        threads.append(t)
15        t.start()
16
17    #ponto sincronismo: só passa para a próxima instrução quando os
18    # todos os threads tiverem chegado aqui
19    for i in threads:
20        i.join()
21
22    print("Finalizou-se o processo")
23
24 ini = time.time() #coleta o tempo inicial
25 exec()
26 fim = time.time() #coleta o tempo Finalizou
27
28 print("Tempo de execução ", fim - ini)
```

Exemplo de coleta de tempo de execução no algoritmo da aula anterior

```
1 import threading
2 import time
3
4 def worker(thread_num):
5     print("Iniciou-se a execução do thread ", thread_num)
6     time.sleep(5)
7     print("Acabou a execução do thread ", thread_num)
8
9 def exec():
10    threads= []
11
12    for i in range(5):
```

Reflita:

- Se você rodar este algoritmo dezenas ou centenas de vezes, o que poderia fazer para agilizar o processo de coleta do tempo médio?
- Você pode omitir todos os tempos intermediários da saída? E se tiver a série possuir pontos discrepantes (*outliers*)?
- Todas essas chamadas de I/O são necessárias para a execução do código? Lembre-se que o tempo de execução delas também será considerado se estiver entre o tempo inicial e final.

```
23
24 ini = time.time() #coleta o tempo inicial
25 exec()
26 fim = time.time() #coleta o tempo Finalizou
27
28 print("Tempo de execução ", fim - ini)
```

# Reflita!

- Sempre reflita sobre as questões abaixo e sua relação com o speedup coletado
  - Qual o hardware usado?
  - Qual o sistema operacional usado?
  - Qual a linguagem e recursos da linguagem usados?
  - Qual o tamanho do conjunto de dados ou do problema que está sendo resolvido?
  - Por que determinada abordagem obteve melhor desempenho em relação a outra?
  - Onde no meu algoritmo pretendo aumentar a vazão com a proposta que fiz?

# Comunicação entre processos (sistemas distribuídos)

Bianca Portes

[bianca.castro@ifsudestemg.edu.br](mailto:bianca.castro@ifsudestemg.edu.br)

# Recordando...

Threads	X	Processos
Possui compartilhamento de memória		Sem compartilhamento de memória
Início/troca de tarefas é computacionalmente menos custosa		Início/troca de tarefas é computacionalmente custosa
Requer poucos recursos		Requer mais recursos computacionais
Necessita de mecanismos de sincronização para manipular dados corretamente		Sincronização de memória não é necessária

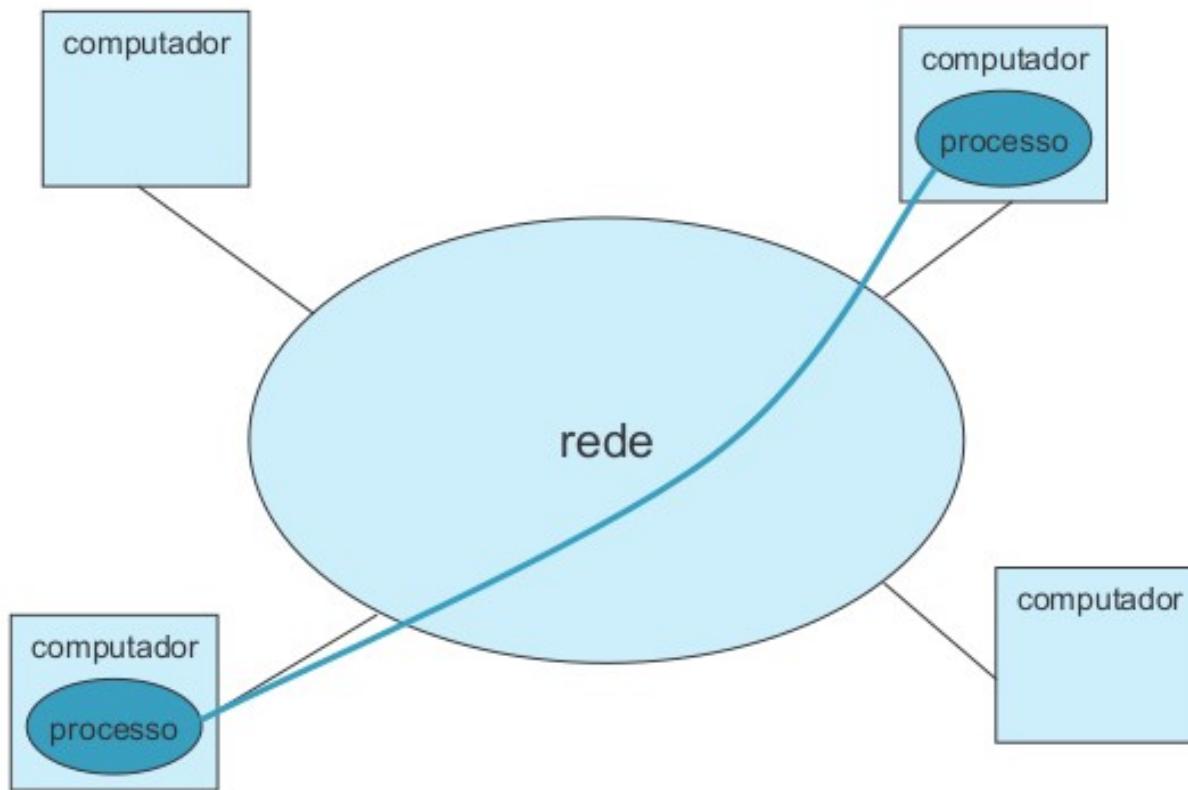
Também aprendemos uma métrica comparativa entre algoritmos, o *speedup*

# Aula de hoje

- A aula de hoje objetiva
  - Recordar os protocolos de comunicação em rede
  - Introduzir os tipos de comunicação em sistemas distribuídos
  - Implementar e analisar um serviço cliente/servidor de exemplo

# Introdução

- Comunicação entre processos está no coração de todo sistema distribuído
- A comunicação em sistemas distribuídos é sempre baseada em **troca de mensagens** de baixo nível entre processos como a oferecida pela rede subjacente



# Modelos de comunicação em sistemas distribuídos

- Troca de mensagens de forma básica:
  - Orientado a mensagens (UDP)
  - Orientado a fluxo (TCP)
  - Comunicação de grupo
- Troca de mensagens em mais alto nível (*via middleware*)
  - *Remote Procedure Call* (RPC)
    - Comunicação cliente-servidor
  - *Remote Method Invocation* (RMI)
    - Comunicação entre objetos distribuídos

**A camada de *middleware* faz a interface entre a camada de aplicação e a camada de transporte!**



**A camada de transporte fornece ferramentas efetivamente utilizadas pela maioria dos sistemas distribuídos**

# Modelos de comunicação em sistemas distribuídos

- Troca de mensagens de forma básica:

- Orientado a mensagens (UDP)
  - Orientado a fluxo (TCP)
  - Comunicação de grupo

Começaremos por aqui... pela descrição de parte dos protocolos do Modelo OSI

- Troca de mensagens em mais alto nível (via *middleware*)

- *Remote Procedure Call* (RPC)
    - Comunicação cliente-servidor
  - *Remote Method Invocation* (RMI)
    - Comunicação entre objetos distribuídos

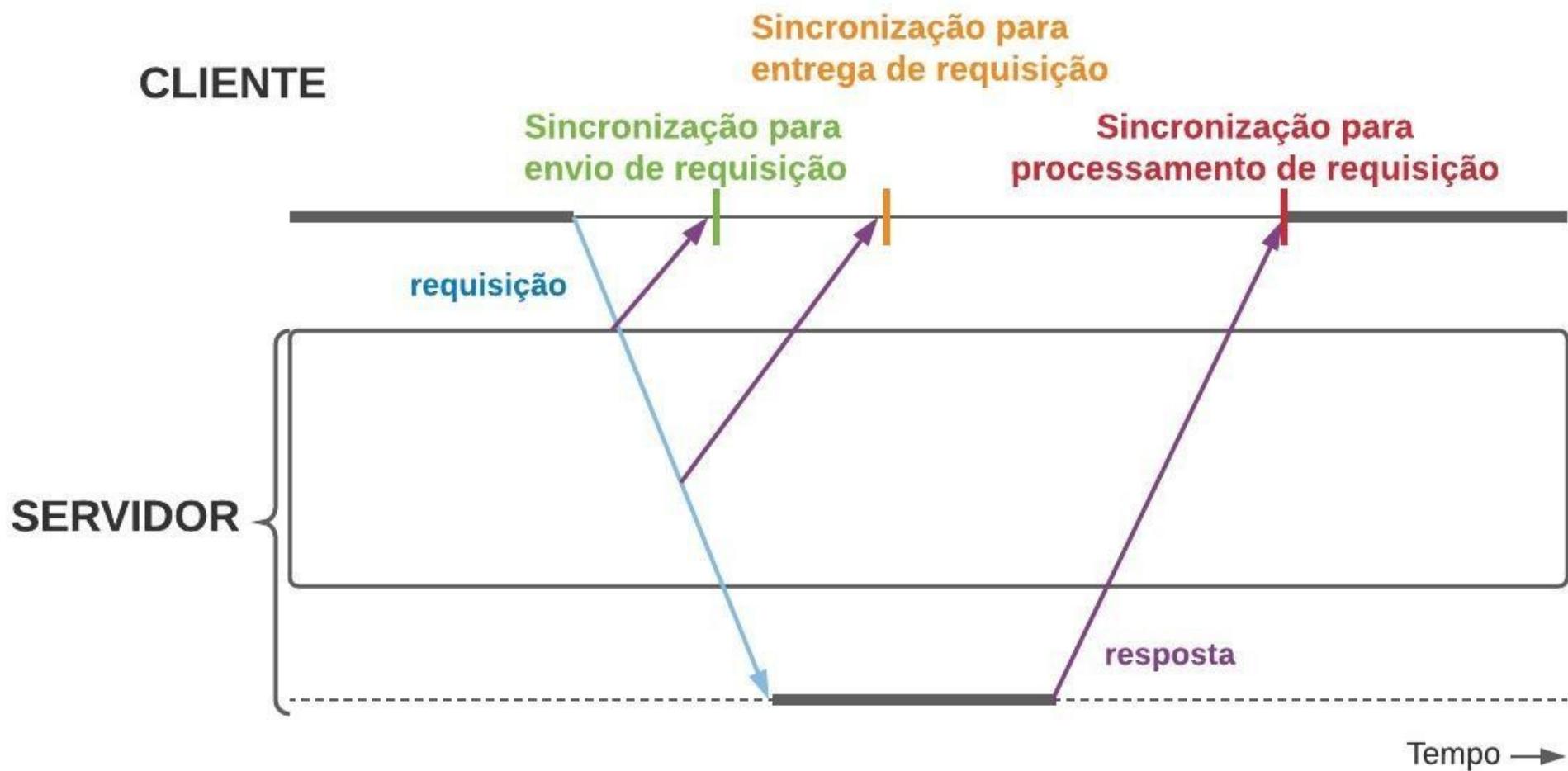
# Revisão modelo básico

- Quais as características da comunicação entre processos que utilizam protocolos UDP? E quais as relacionadas ao TCP?
- O que é a comunicação em grupo?

# Características da comunicação entre processos

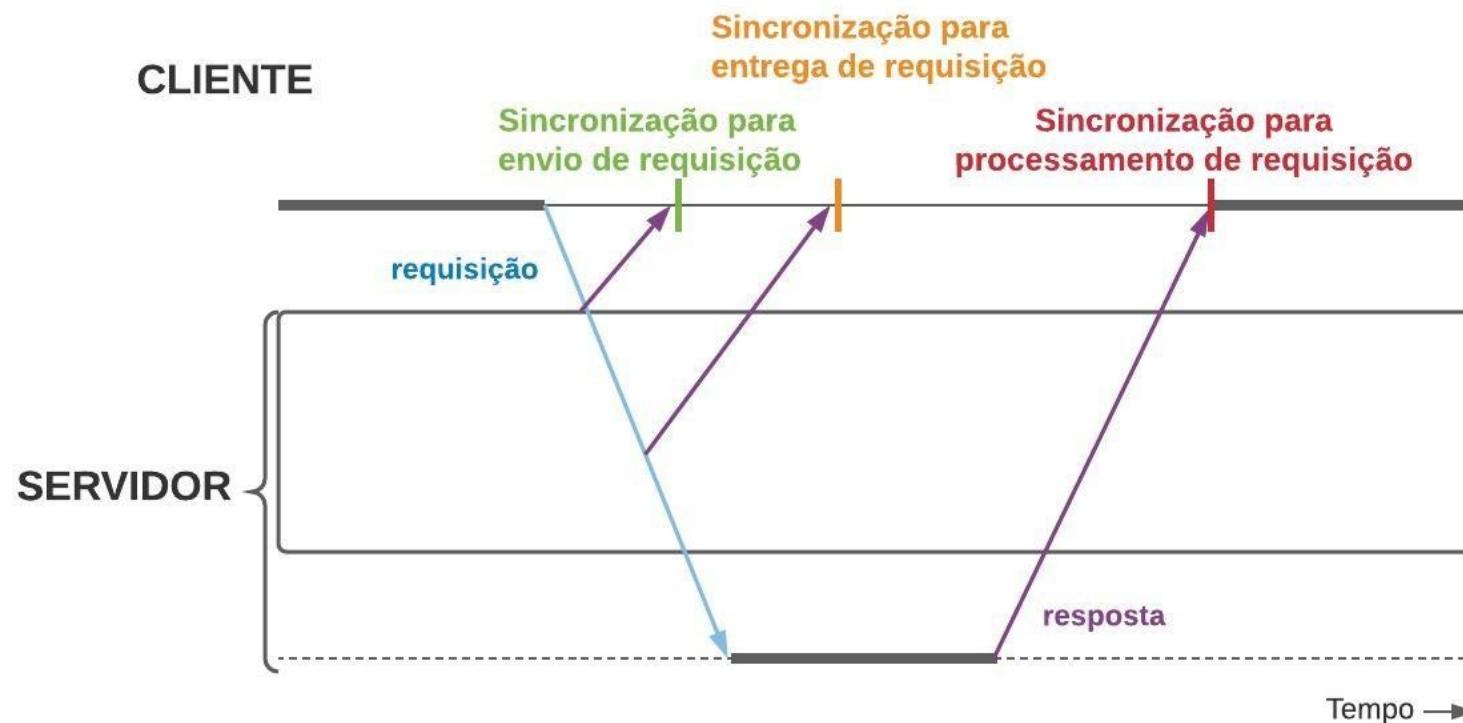
- A passagem de mensagens entre um par de processos é feita a partir duas operações de comunicação: *send* e *receive*
- Esta atividade pode implicar na sincronização dos dois processos

# Tipos de comunicação em SD



- a) comunicação persistente vs. transiente
- b) comunicação síncrona vs. assíncrona

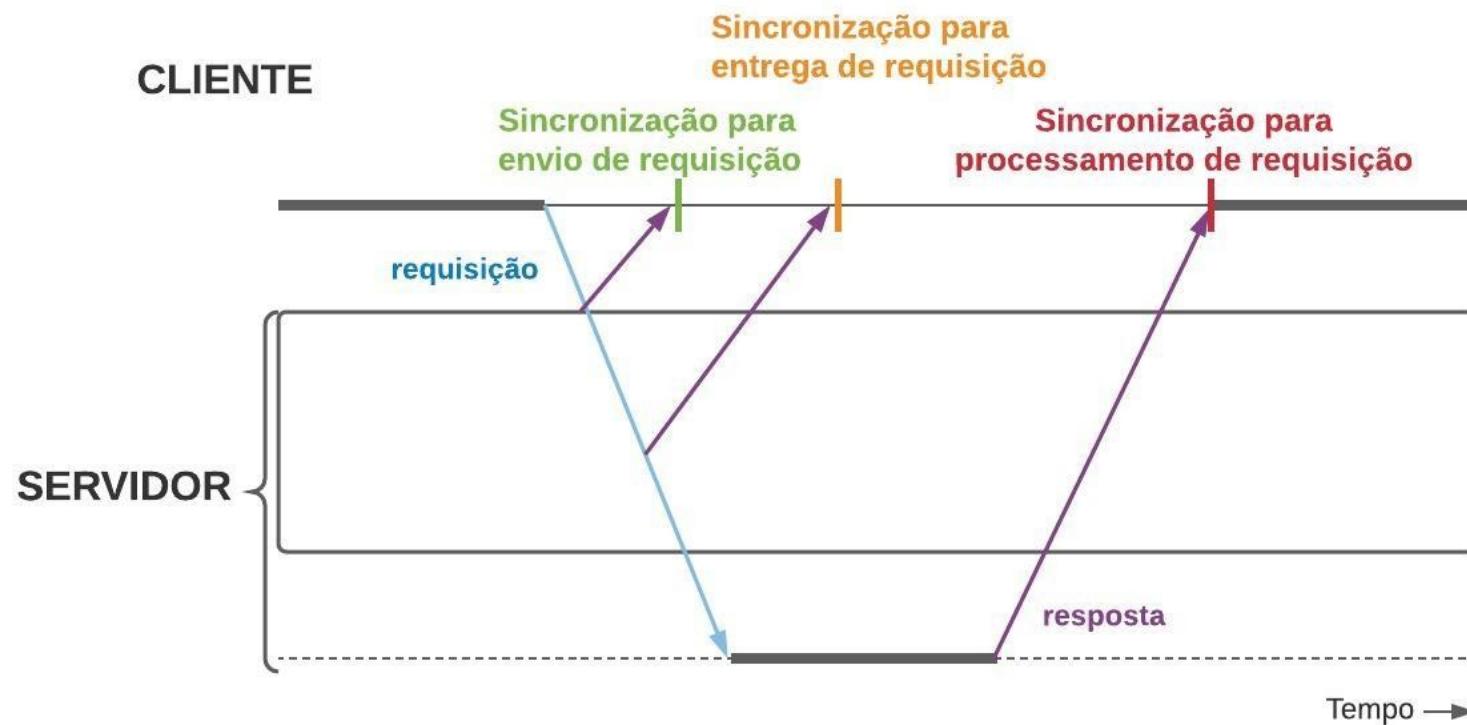
# Persistente vs. transiente



Transiência e persistência referem-se ao comportamento da serviço:

- **Comunicação transiente:** destinatário não estando disponível no momento da entrega da mensagem, a informação é perdida
- **Comunicação persistente:** a mensagem é guardada pelo tempo que for necessário até que seja possível entregar ao destinatário

# Síncrona vs. assíncrona



- Comunicação síncrona: processos remetente e destino são sincronizados a cada mensagem

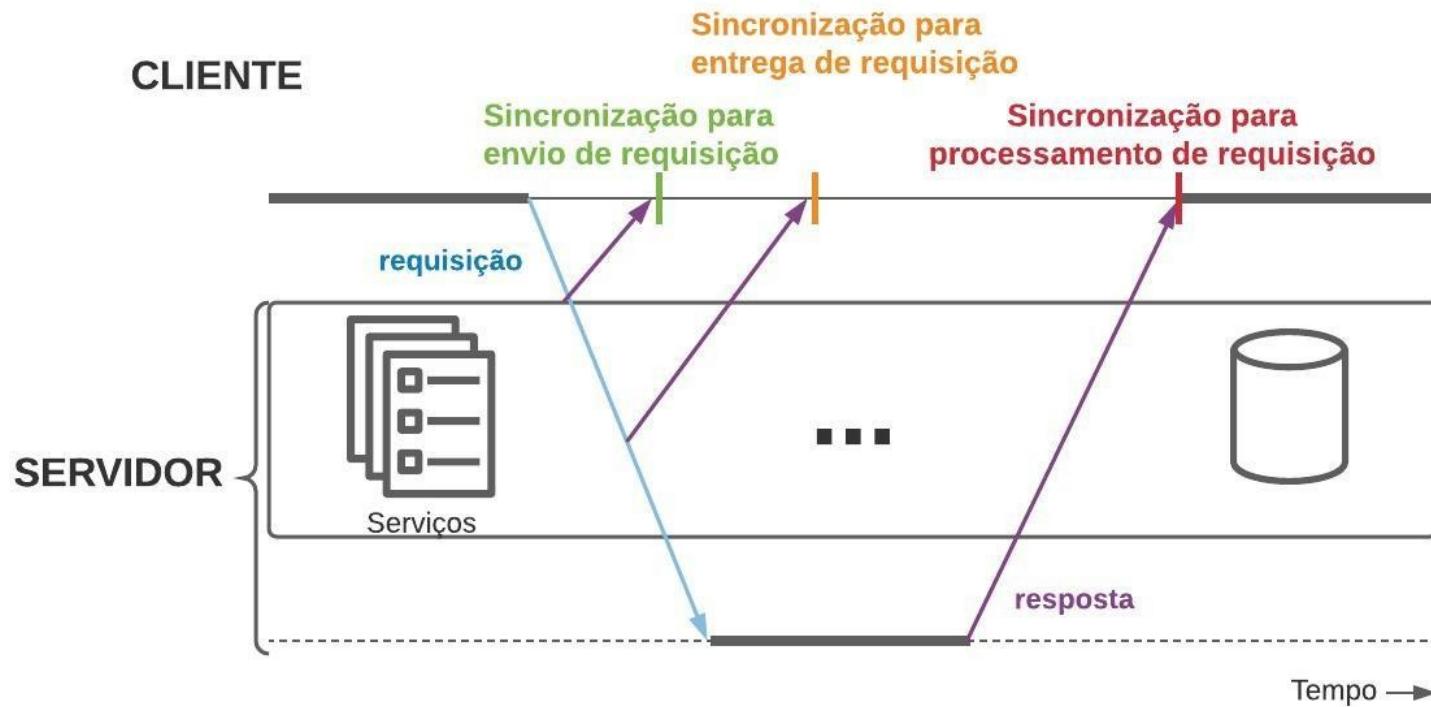
Pontos de sincronização: remetente bloqueado

- no envio da requisição
- na entrega da requisição
- no processamento da requisição

# Síncrona vs. assíncrona

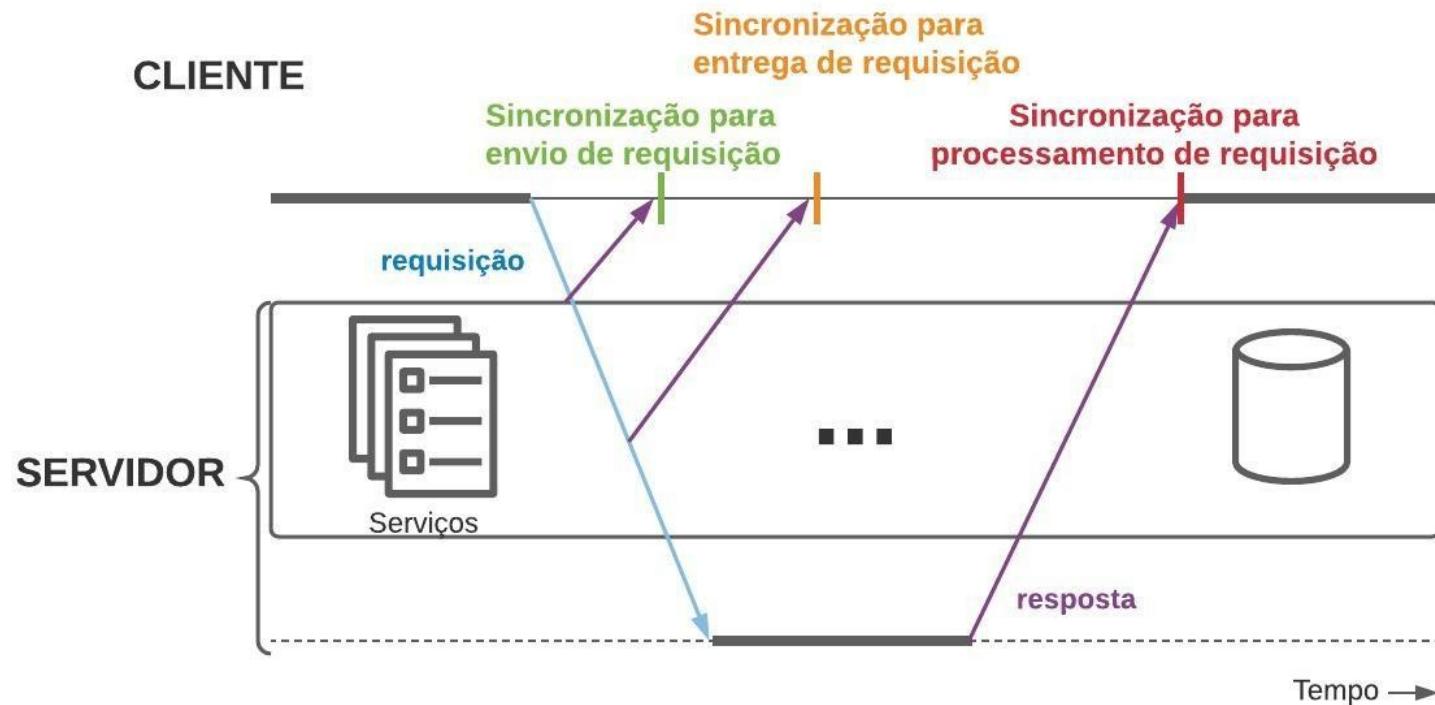
- Comunicação assíncrona
  - Processos remetente não fica bloqueado
    - mensagem é copiada para um buffer local e o processo remetente continua seu processamento enquanto a transmissão ocorre
  - Processo destino pode ser
    - blockante: processo bloqueado enquanto a mensagem não chega
    - não-blockante: processo continua execução, podendo ser notificado (p.ex. por uma interrupção) da chegada de uma mensagem

# Exemplo 1: cliente/servidor



- A arquitetura cliente/servidor normalmente é baseada no modelo de **comunicação síncrona e transient**
  - cliente e servidor devem estar ativos no momento da comunicação
  - cliente envia a mensagem e fica bloqueado até obter sua resposta
  - servidor apenas aguarda requisições e as processa

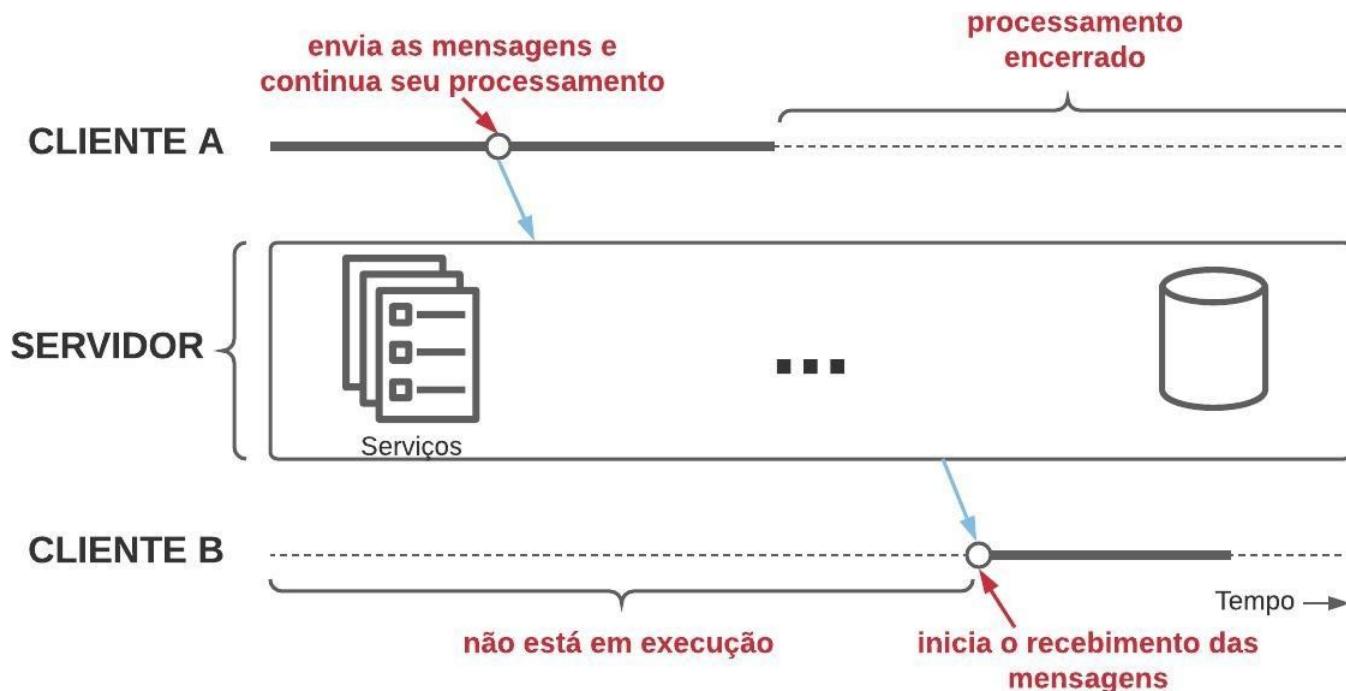
# Exemplo 1: cliente/servidor



Problemas da comunicação síncrona?

- o cliente fica parado enquanto estiver esperando uma resposta
- falhas devem ser tratadas com rapidez (o cliente está esperando)
- há serviços que não se encaixam neste modelo (ex.: email)

# Ex.2: Microsserviço



A arquitetura de microsserviços é frequentemente baseada no modelo de **comunicação assíncrona e persistente** (utilização de filas)

- processos trocam mensagens entre si
- o remetente não precisa esperar por uma resposta imediata, pode fazer outras coisas enquanto espera
- normalmente assegura tolerância a falhas

# LEMBRANDO: comunicação via sockets

- Em protocolos Internet as mensagens são enviadas para destinos definidos pelo par endereço IP e porta local
  - Transmissão entre o socket de um processo e o socket de outro processo
  - Se nenhum processo tiver soquete associado à porta de destino, as mensagens são descartadas

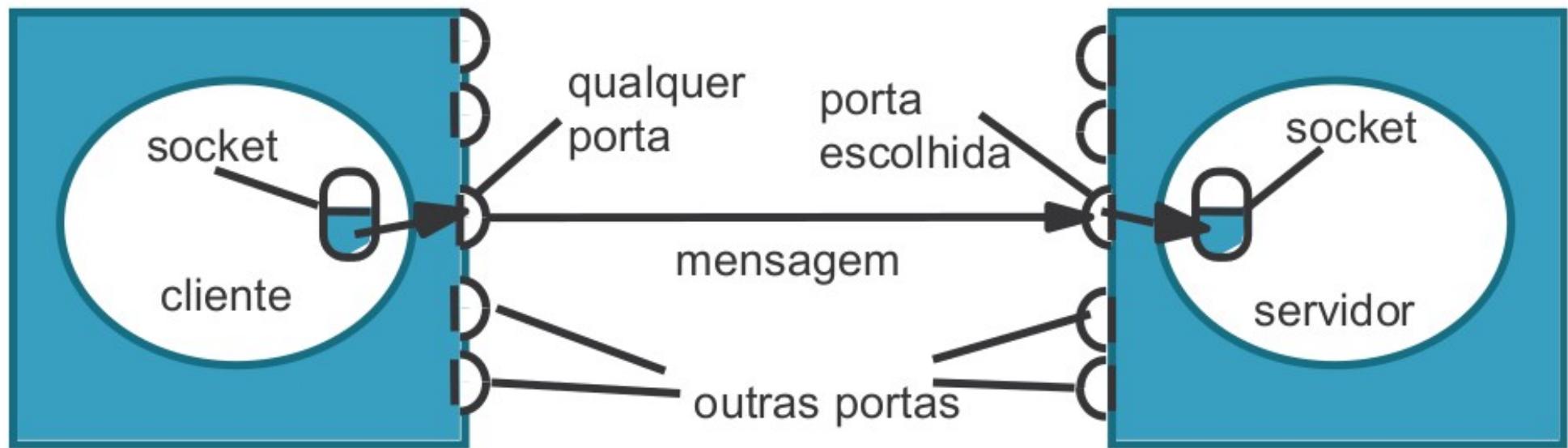
# Exemplo troca de mensagens básico (via socket)

- A porta local é um destino de mensagem dentro de um computador, especificado como um valor inteiro
  - Cada computador tem  $2^{16}$  portas locais
  - Processo não pode compartilhar uma porta com outros processos (exceção multicast)
  - Processos podem usar várias portas para receber mensagens

Comunicação entre processos consiste em transmitir uma mensagem entre um soquete de um processo e um soquete de outro processo

# Comunicação via sockets

Exemplo de uma **comunicação transiente assíncrona**



endereço IP = **192.168.0.106**

endereço IP = **192.168.0.107**

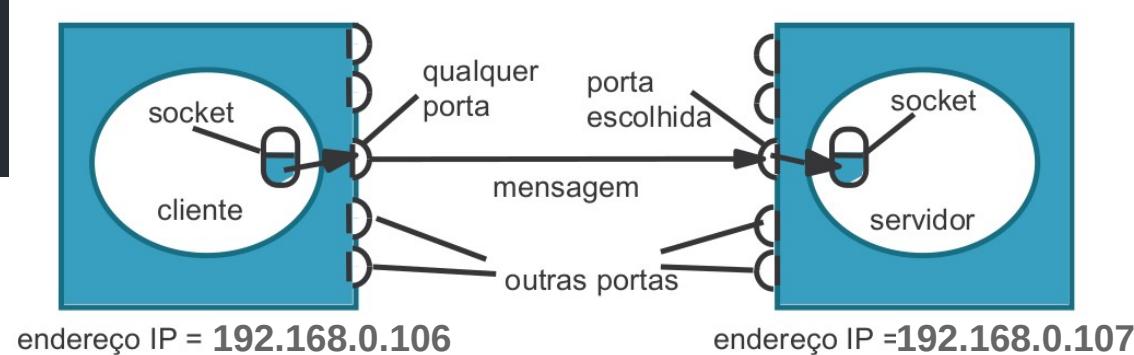
protocolo de transporte: UDP

Que porta usar? Procure uma porta que não esteja em uso por outro processo  
O comando abaixo listará todas as portas em uso no momento:

```
$ netstat -o -n -a
```

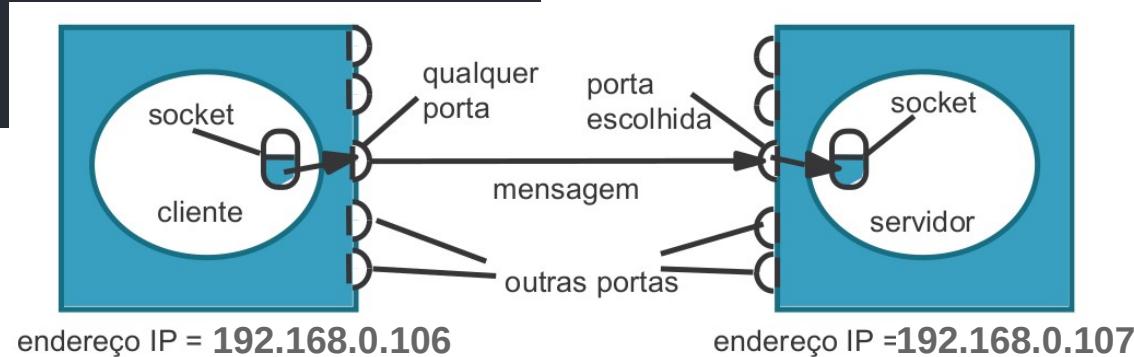
# Implementando o cliente

```
1 import socket  
2  
3 HOST = '192.168.0.107'      # Endereco IP do Servidor  
4 PORT = 5005                  # Porta que o Servidor esta  
5 BUFFER_SIZE = 12000          # Tamanho em bytes do buffer de recepcao  
6  
7 dest = (HOST, PORT)  
8  
9 clientSocket = socket.socket(socket.AF_INET,           #Internet  
10                         socket.SOCK_DGRAM)           # UDP  
11  
12 print("Para sair use CTRL+X\n")  
13 msg = input()  
14 try:  
15     while msg != '\x18':  
16         clientSocket.sendto (msg.encode("utf-8"), dest)  
17         msg = input()  
18  
19 finally:  
20     clientSocket.close()
```



# Implementando o servidor

```
1 import socket  
2  
3 HOST = '192.168.0.107'          # Endereco IP do Servidor  
4 PORT = 5005                      # Porta que o Servidor esta escutando  
5 BUFFER_SIZE = 12000                # Tamanho em bytes do buffer de recepcao  
6  
7 serverSocket = socket.socket(socket.AF_INET,      #Internet  
8                               socket.SOCK_DGRAM)    # UDP  
9  
10 orig = (HOST, PORT)  
11 serverSocket.bind(orig)  
12  
13 try:  
14     while True:  
15         msg, cliente = serverSocket.recvfrom(BUFFER_SIZE)  
16         print(cliente, msg.decode('utf-8'))  
17 finally:  
18     serverSocket.close()  
19     print("\n\n\tConexao finalizada\n\n")
```



# API Sockets: TCP/IP

Primitva	Significado
socket	Cria um ponto final de conexão
bind	Associa um endereço local ao socket
listen	Anuncia desejo em aceitar conexões
accept	Bloqueia o chamador até que chegue um pedido de conexão
connect	Tentaativamente estabelecer uma conexão
send	Envia dados
receive	Recebe dados
close	Termina a conexão