

# MCTA025-13 - SISTEMAS DISTRIBUÍDOS

PROCESSOS, VIRTUALIZAÇÃO E MIGRAÇÃO

---

Emilio Francesquini

25 de junho de 2018

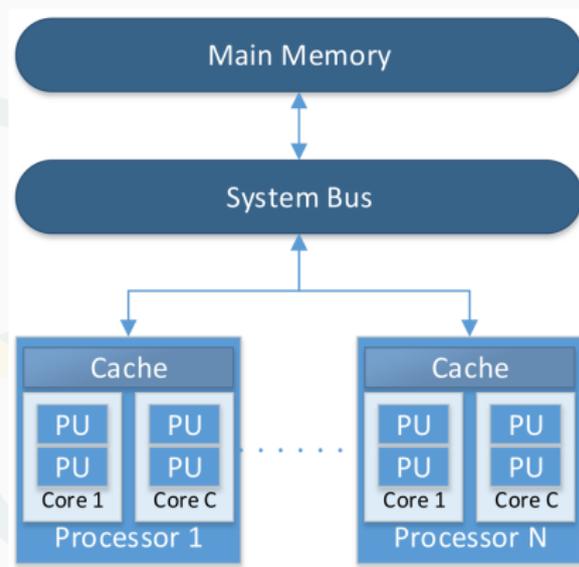
Centro de Matemática, Computação e Cognição  
Universidade Federal do ABC



- Estes slides foram preparados para o curso de **Sistemas Distribuídos na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram adaptados daqueles originalmente preparados (e gentilmente cedidos) pelo professor **Daniel Cordeiro, da EACH-USP** que por sua vez foram baseados naqueles disponibilizados online pelos autores do livro “Distributed Systems”, 3ª Edição em:  
<https://www.distributed-systems.net>.

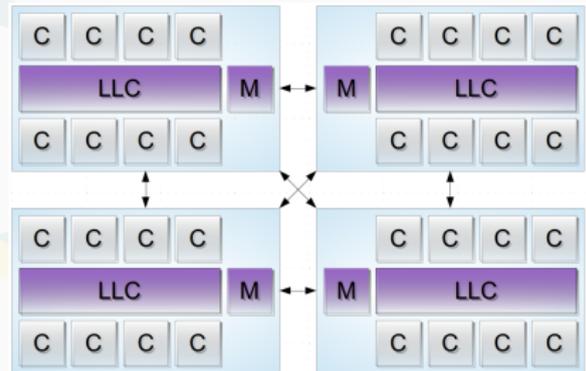
# COMPUTADORES DE MEMÓRIA COMPARTILHADA

- Todas as posições de memória podem ser acessadas por todos as unidades de processamento
  - Espaço de endereçamento único
- Symmetric Multiprocessing Platforms (SMP) são arquiteturas de memória compartilhada que possuem ao menos duas unidades de processamento
- UMA - Uniform Memory Access

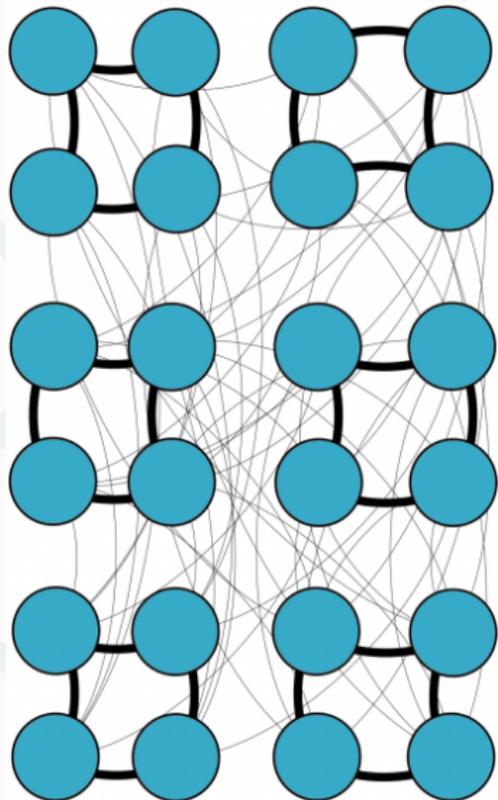


# ARQUITETURAS NUMA

- NUMA – Non-Uniform Memory Access
- Pode-se imaginar uma máquina NUMA como sendo um conjunto de máquinas UMA ligadas por uma rede de interconexão de alto desempenho
- Espaço de endereçamento único
  - Desempenho para acesso à memória variável



- Topologias comuns
  - **Toroidal**: Kalray MPPA-256
  - **Grade**: Tiler TILE-GX\*
  - **Anel**: Intel Xeon Phi
  - **Hipercubo**: SGI Altix UV2000
- **Conhecimento da topologia é essencial** para desenvolver programas com um **bom desempenho**



SGI Altix UV2000 – Hipercubo (5D) com conexões privilegiadas 4/48

## Ideia básica

Construir um **processador virtual** com software, em cima dos processadores físicos:

**processador:** (hardware) provê um conjunto de instruções junto com a capacidade de executá-las automaticamente

**thread:** (software) um processador mínimo com um **contexto** que possui uma série de instruções que podem ser executados. Gravar o contexto de um thread implica parar a execução e guardar todos os dados necessários para continuar a execução posteriormente

**processo:** (software) um processador em cujo contexto pode ser executado uma ou mais threads. Executar um thread significa executar uma série de instruções no contexto daquele processo.

- Uma das abstrações mais importantes de um SO
  - Representam a **execução** de um programa
  - Execuções simultâneas de um programa são representadas por diversos processos
- Por segurança, os **espaços de memória** de cada processo são **isolados**
- Evita problemas que seriam causados por ataques deliberados ou por bugs
- Cada processo é uma **linha de execução independente** escalonada pelo SO

- Frequentemente um mesmo processo necessita fazer **mais de uma coisa por vez**. Ex.: Navegador de Internet
- Da mesma maneira que processos fornecem múltiplas linhas de execução em uma máquina, threads permitem **múltiplas linhas de execução em um só processo**
- Como efetivamente todos os threads são o mesmo processo, **todos têm acesso ao mesmo espaço de memória** e a todos os recursos disponíveis a este processo

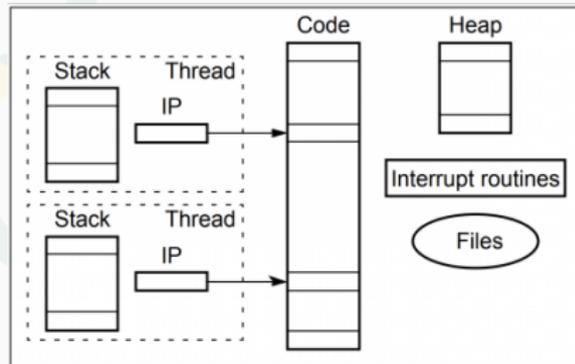
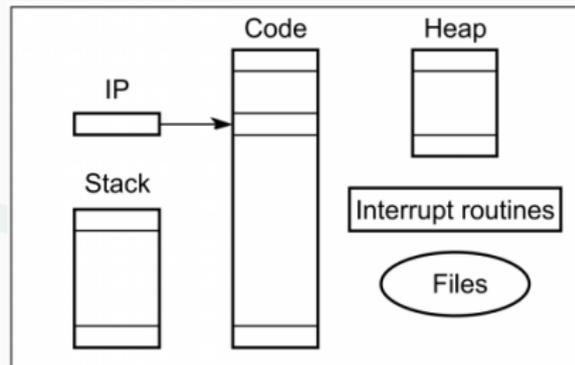
# PROCESSOS VS. THREADS

- **Processos**

- Separação total dos programas: pilhas, espaço de memória, descritores de arquivos, tabelas de páginas...

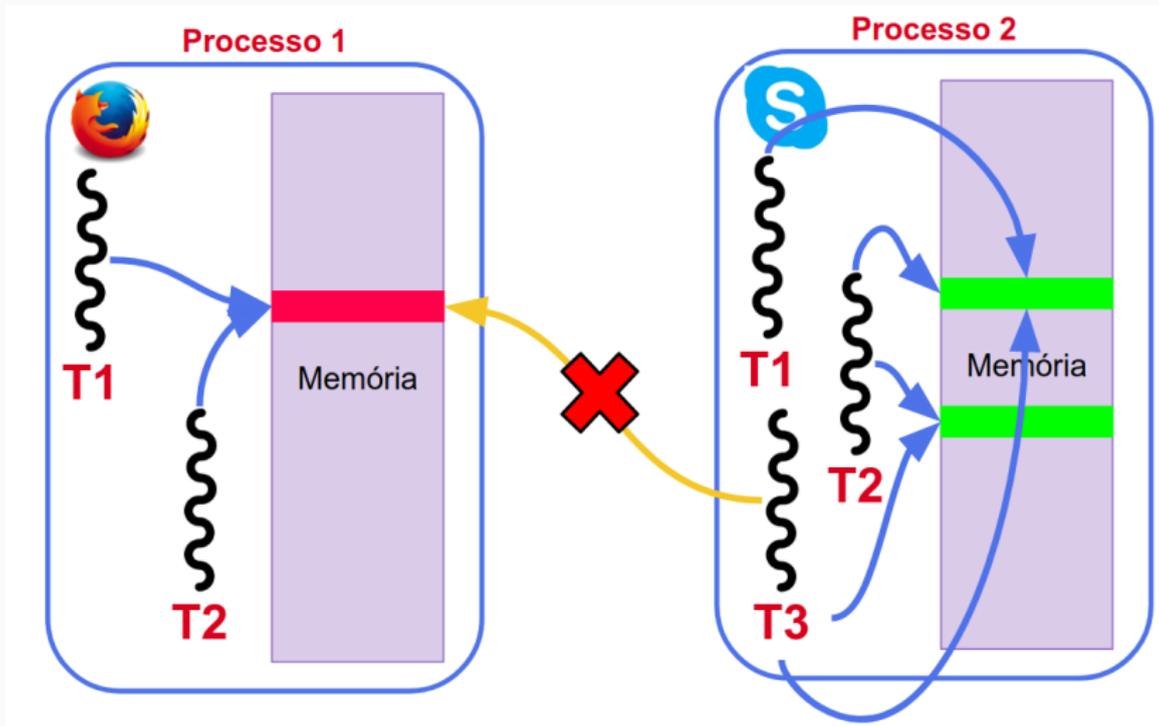
- **Threads**

- Compartilham o espaço de memória, descritores de arquivos, tabelas de páginas,...
- Uma pilha e IP por thread

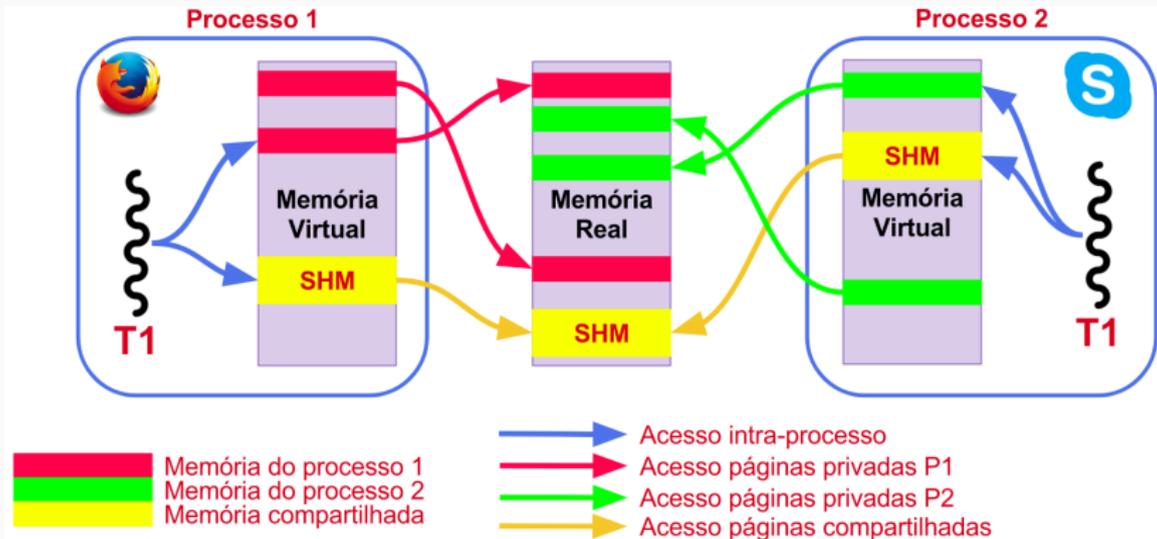


Parallel Programming Techniques & Applications Using Networked Workstations & Parallel Computers 2nd ed., by B. Wilkinson & M. Allen

# PROCESSOS VS. THREADS - VERSÃO SIMPLIFICADA



# PROCESSOS VS. THREADS - VERSÃO (MENOS) SIMPLIFICADA



# PROCESSOS VS. THREADS

- SO provê mecanismos para dividir o tempo do processador entre processos e threads, escalonando-os nas unidades de processamento disponíveis
  - Bloqueios por alguma operação de E/S causam uma troca do processo/thread pelo próximo na fila
- Trocas de contexto entre threads são **baratas**
  - Basta trocar o IP e mais alguns registradores
- Trocas de contexto entre processos são **mais caras**
  - Exigem troca da tabela de páginas, troca de IP, troca de rotinas de manuseio de interrupções, ...
- Ainda assim há momentos que o uso de processos pode ser preferível. Exemplo: Google Chrome

## Contextos

**Contexto do processador:** um conjunto mínimo de valores guardados nos registradores do processador, usado para a execução de uma série de instruções (ex: ponteiro de pilha, registradores, contador de programa, etc.)

**Contexto de thread:** um conjunto mínimo de valores guardado em registradores e memória, usado para a execução de uma série de instruções (i.e., contexto do processador e estado)

**Contexto de processo:** um conjunto mínimo de valores guardados em registradores e memória, usados para a execução de uma thread (i.e., contexto de threads e os valores dos registradores de MMU — Memory Management Unit)

Observações:

1. Threads compartilham o mesmo espaço de endereçamento. A realização da troca de contexto pode ser feita independentemente do sistema operacional
2. A troca de processos é mais custosa, já que envolve o sistema operacional
3. Criar e destruir threads é muito mais barato do que fazer isso com processos

## Problema:

O núcleo do sistema operacional deve prover threads, ou elas devem ser implementadas em nível de usuário?

## Solução no nível de usuário

- Todas as operações podem ser realizadas **dentro de um único processo** — **muito** mais eficiente
- Todos os serviços providos pelo núcleo são feitos *em nome do processo na qual a thread reside* — se o núcleo decidir bloquear a thread, o processo inteiro será bloqueado
- Threads são usadas quando há muitos eventos externos: *threads são bloqueadas com base nos eventos recebidos* — se o kernel não puder distinguir as threads, como permitir a emissão de sinais do SO para elas?

## Solução no nível do SO

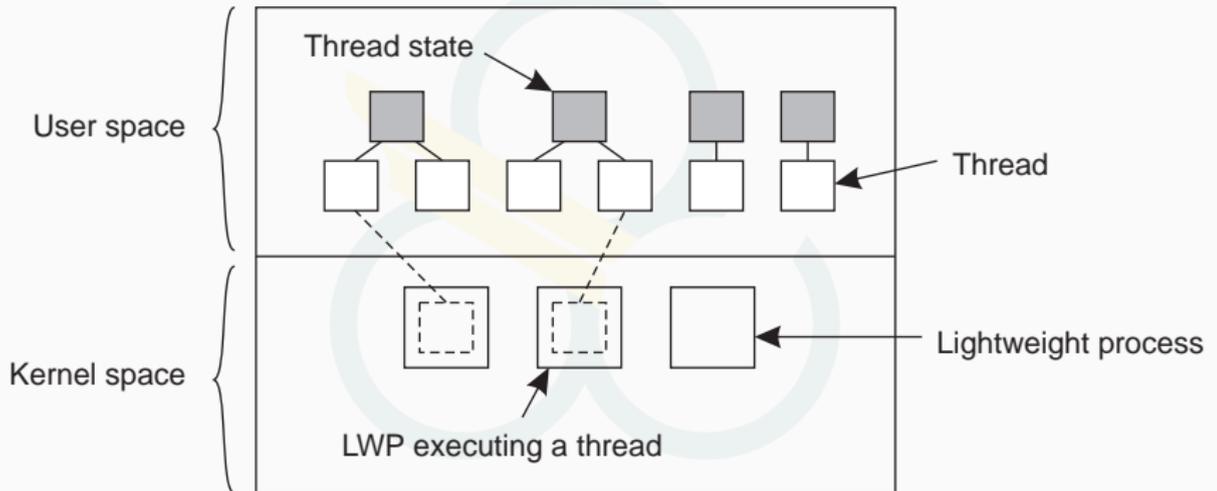
- O núcleo contém a implementação do software de threading. *Todas* as operações são chamadas de sistemas
- Operações que bloqueiam uma thread não são mais um problema: o núcleo escalona outra thread ociosa dentro do mesmo processo
- Tratamento de eventos externos mais simples: o núcleo (que recebe todos os eventos) escalona a thread associada com aquele evento
- O problema é a perda de eficiência devido ao fato de que todas as operações em threads requerem um *trap* pro núcleo

## Conclusão

O melhor é tentar juntar threads de nível de usuário e de nível do SO em um único conceito.

# THREADS DO SOLARIS

Introduz uma abordagem em dois níveis para threads: **processos leves** que podem ser executar threads de nível de usuário



## Operação principal

- uma thread de nível de usuário realizam uma chamadas de sistema: o LWP (*light-weight process*) que estiver executando aquela thread bloqueia. A thread continua associada àquele LWP.
- O núcleo pode escalonar outro LWP com uma thread associada pronta para execução. Essa thread pode ser trocada por qualquer outra thread de nível de usuário que esteja pronta.
- Uma thread executa uma operação de nível de usuário bloqueante — faça troca de contexto para uma thread pronta (e então a associe ao mesmo LWP).
- Quando não há threads para executar, um LWP pode ficar ocioso, e mesmo destruído pelo núcleo.

## Clientes web multithreaded – escondem a latência da rede:

- Navegador analisa a página HTML sendo recebida e descobre que *muitos outros arquivos devem ser baixados*.
- Cada arquivo é baixado por uma thread separada; cada uma realiza uma requisição HTTP (bloqueante)
- A medida que os arquivos chegam, o navegador os exibem.

## Múltiplas chamadas requisição-resposta (RPC) para outras máquinas

- Um cliente faz várias chamadas simultâneas, cada uma em uma thread diferente
- Ele espera até que todos os resultados tenham chegado.
- Obs: se as chamadas são a servidores diferentes, você pode ter um **speed-up linear**

## Melhorias no desempenho

- Iniciar uma thread é **muito** mais barato do que iniciar um novo processo
- Ter servidores single-threaded impedem o uso de sistemas multiprocessados
- Tal como os clientes: **esconda a latência da rede** reagindo à próxima requisição enquanto a anterior está enviando sua resposta.

## Melhorias na estrutura

- A maioria dos servidores faz muita E/S. Usar chamadas bloqueantes simples e bem conhecidas simplifica o programa.
- Programas multithreaded tendem a ser menores e mais fáceis de entender, já que simplificam o fluxo de controle.

VIRTUALIZAÇÃO

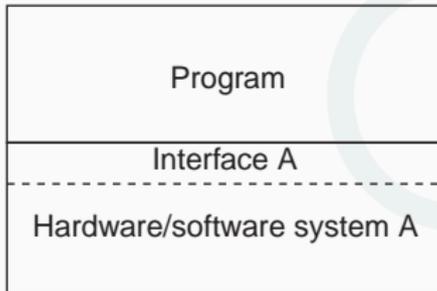
---



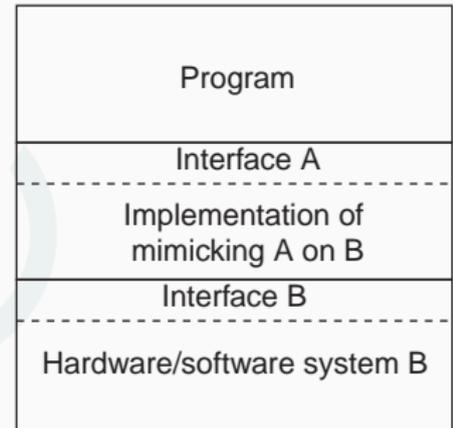
# VIRTUALIZAÇÃO

É cada vez mais importante:

- Hardware **muda mais rápido** do que software
- Melhora a **portabilidade** e a migração de código
- Provê **isolamento** de componentes com falhas ou sendo atacados

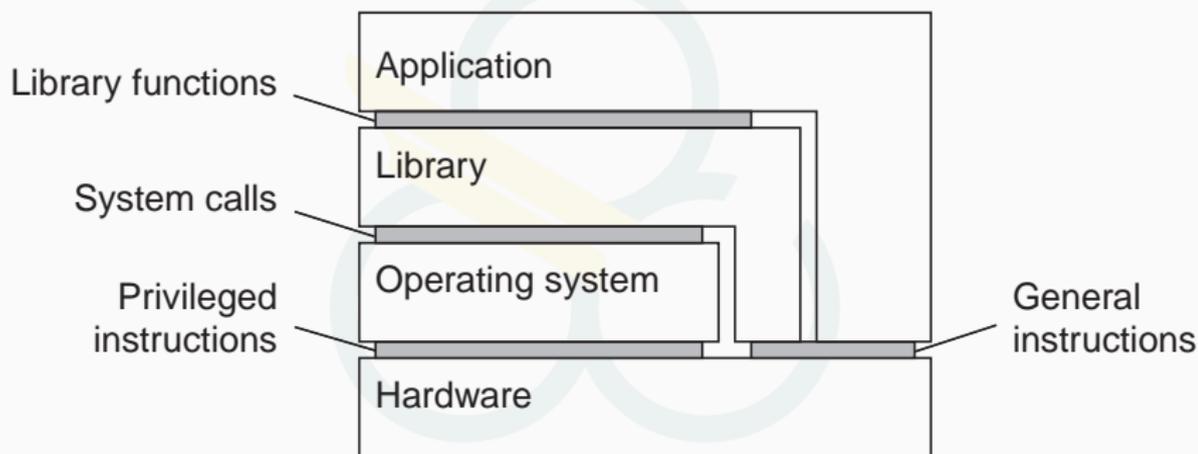


(a)

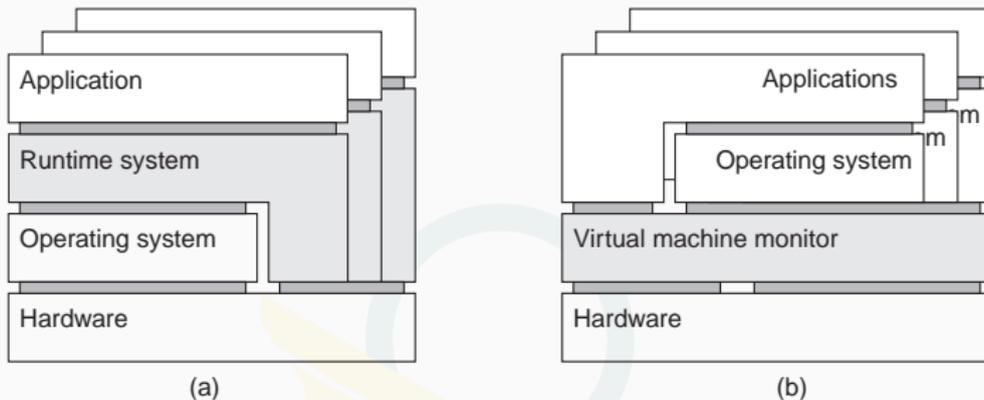


(b)

Virtualização pode ocorrer em diferentes níveis, dependendo das **interfaces** oferecidas pelos diferentes componentes do sistema:



## PROCESSOS VMS VS. MONITORES DE VM



- **Processos VMs:** um programa é compilado para um código intermediário (portátil) que é executado por um interpretador. Ex: Java VM.
- **Monitor de VM:** uma camada de software que imita o conjunto de instruções do hardware — pode executar um sistema operacional completo e suas aplicações. Ex: VMware, VirtualBox.

Monitores de VM são executadas em cima de sistemas operacionais existentes.

- Realizam **tradução binária**: enquanto executam uma aplicação ou sistema operacional, traduzem as instruções para as instruções da máquina física
- Distinguem **instruções sensíveis**: traps para o núcleo original (system calls ou instruções privilegiadas)
- Instruções sensíveis são substituídas por chamadas ao Monitor de VM

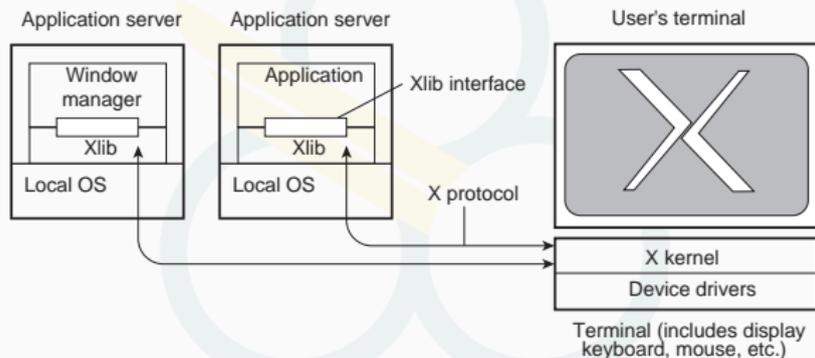
# ARQUITETURAS E PROCESSOS DE SISTEMAS DISTRIBUÍDOS

---

CLIENTES

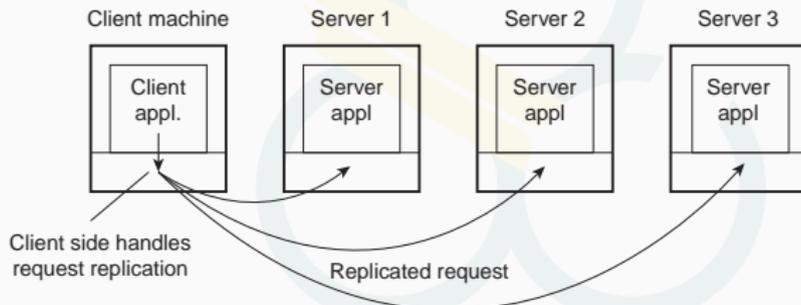
## CLIENTES: INTERFACES DE USUÁRIOS

A maior parte dos softwares do lado do cliente é especializada em interfaces (gráficas) de usuário. O *X protocol* é um exemplo de *thin-client network computing*.



## Geralmente adaptado para transparência de distribuição

- **transparência de acesso:** *stubs* do cliente para RPC
- **transparência de localização/migração:** deixe o software cliente manter o controle sobre a localização atual
- **transparência de replicação:** múltiplas evocações são gerenciadas pelo stub do cliente:



- **transparência de falhas:** podem geralmente ser responsabilidade só do cliente (que tenta esconder falhas de comunicação e do servidor)

# SERVIDORES

## Modelo básico

Um processo que implementa um serviço específico em nome de uma coleção de clientes. Ele espera pela requisição de um cliente, garante que a requisição será tratada e, em seguida, passa a esperar pela próxima requisição.

Dois tipos básicos:

**Servidores iterativos** o servidor trata uma requisição antes de atender a próxima

**Servidores concorrentes** usa um despachante (*dispatcher*), que pega uma requisição e repassa seu tratamento a uma *thread*/processo separado

## Observação

É mais comum encontrarmos servidores concorrentes: eles podem tratar múltiplas requisições mais facilmente, principalmente se for necessário realizar operações bloqueantes (em discos ou outros servidores).

## SERVIDORES: ORGANIZAÇÃO GERAL

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol
sunrpc	111	SUN RPC (portmapper)

Cada requisição a uma porta é atribuída a um processo dinamicamente, via *superservers* (processo que inicia subprocesso para tratar a requisição; ex: UNIX *inetd*) ou *daemons* (processos que se registram em uma porta).

## Problema:

É possível **interromper** um servidor uma vez que ele já tiver aceito (ou estiver processando) uma requisição de serviço?

## Solução 1: usar uma porta diferente para dados urgentes

- O servidor mantém uma thread/processo separado para mensagens urgentes
- Se uma mensagem urgente chegar, a requisição associada é colocada em espera
- É necessário que o SO ofereça escalonamento por prioridade

## Problema:

É possível **interromper** um servidor uma vez que ele já tiver aceito (ou estiver processando) uma requisição de serviço?

## Solução 1: usar uma porta diferente para dados urgentes

- O servidor mantém uma thread/processo separado para mensagens urgentes
- Se uma mensagem urgente chegar, a requisição associada é colocada em espera
- É necessário que o SO ofereça escalonamento por prioridade

## Solução 2: usar comunicação de controle da camada de transporte

- TCP permite o envio de mensagens urgentes na mesma conexão
- Mensagens urgentes podem ser recebidas usando tratamento de sinais do SO

### **Servidores *stateless***

Não mantêm informação exata sobre o status de um cliente após ter processado uma requisição:

- Não guarda se um arquivo foi aberto (simplesmente fecha-o e abre de novo se necessário)
- Não promete invalidar o cache do cliente
- Não rastreia os seus clientes

## Servidores *stateless*

Não mantêm informação exata sobre o status de um cliente após ter processado uma requisição:

- Não guarda se um arquivo foi aberto (simplesmente fecha-o e abre de novo se necessário)
- Não promete invalidar o cache do cliente
- Não rastreia os seus clientes

## Consequências

- Clientes e servidores são completamente independentes
- **Inconsistências de estado** devido a problemas no cliente ou servidor são reduzidas
- Possível **perda de desempenho**. Um servidor não pode antecipar o comportamento do cliente (ex: *prefetching*)

### O uso de comunicação orientada a conexão viola o modelo stateless?

O uso de conexões com estado não violam o fato de que os servidores não guardam estado.

Mas é necessário ter em mente que a camada de transporte, sim, mantém estado. Melhor seria usar um protocolo stateless combinado com operações idempotentes<sup>1</sup>.

---

<sup>1</sup>A idempotência é a propriedade que algumas operações têm de poderem ser aplicadas várias vezes sem que o valor do resultado se altere após a aplicação inicial.

## Servidores com estado (*stateful*)

Guardam o status de seus clientes:

- Registram quando um arquivo foi aberto para realização de *prefetching*
- Sabem quando o cliente possui cache dos dados e permitem que os clientes mantenham cópias locais de dados compartilhados

## Servidores com estado (*stateful*)

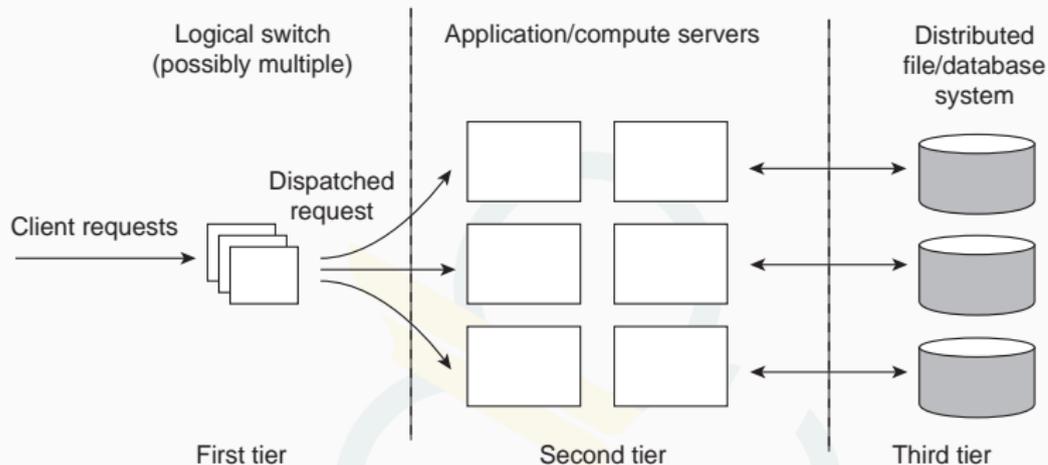
Guardam o status de seus clientes:

- Registram quando um arquivo foi aberto para realização de *prefetching*
- Sabem quando o cliente possui cache dos dados e permitem que os clientes mantenham cópias locais de dados compartilhados

### Observação:

O desempenho de servidores *stateful* pode ser extremamente alto, desde que seja permitido que os clientes mantenham cópias locais dos dados. Nesses casos, **confiabilidade não é o maior problema.**

# AGLOMERADOS DE SERVIDORES: TRÊS CAMADAS DIFERENTES



## Elemento crucial

A primeira camada é responsável por repassar as requisições para um servidor apropriado.

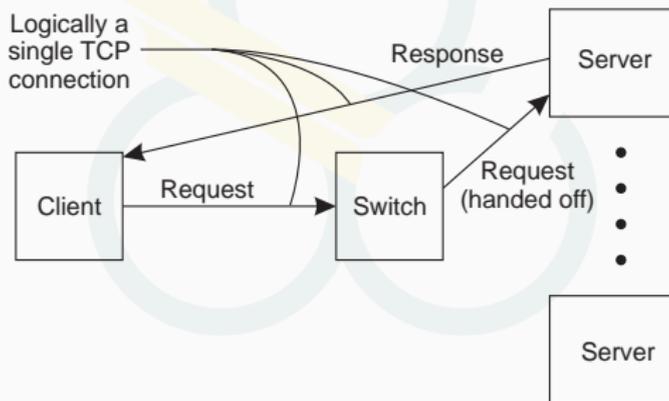
# TRATAMENTO DE REQUISIÇÕES

## Observação:

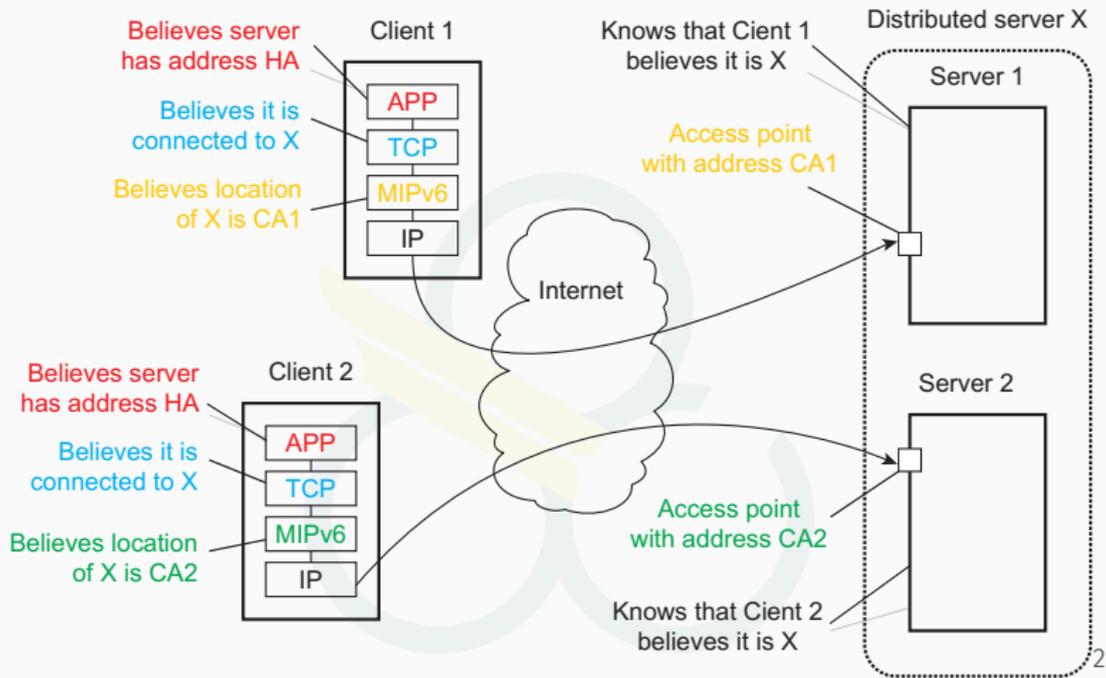
Ter uma única camada tratando toda a comunicação de/para o aglomerado pode levar a criação de um **gargalo**.

## Solução:

Várias, mas uma popular é o chamado **TCP-handoff**:



# SERVIDORES DISTRIBUÍDOS COM ENDEREÇOS IPV6 ESTÁVEIS



<sup>2</sup>MIPv6 = Mobile IPV6; HA = Home Address; CA = Care-of Address

Clientes com Mobile IPv6 podem criar conexões com qualquer outro par de forma transparente:

- Cliente *C* configura uma conexão IPv6 para o **home address** *HA*.
- *HA* é mantido (no nível da rede) por um **home agent**, que repassa a conexão para um endereço **care-of CA** registrado
- *C* aplica uma **otimização de rota** ao encaminhar os pacotes diretamente para o endereço do *CA*, sem passar pelo *home agent*.

Clientes com Mobile IPv6 podem criar conexões com qualquer outro par de forma transparente:

- Cliente C configura uma conexão IPv6 para o **home address HA**.
- HA é mantido (no nível da rede) por um **home agent**, que repassa a conexão para um endereço **care-of CA** registrado
- C aplica uma **otimização de rota** ao encaminhar os pacotes diretamente para o endereço do CA, sem passar pelo *home agent*.

## CDNs colaborativas

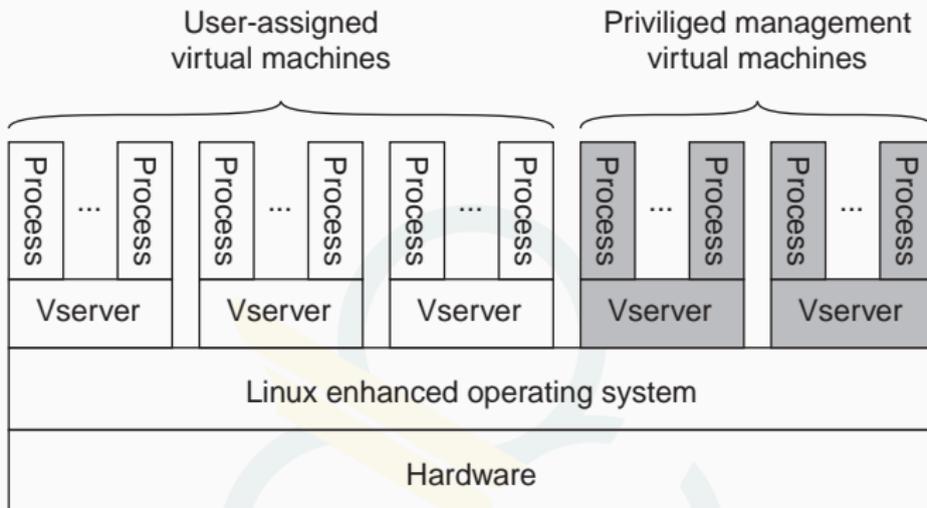
O servidor original mantém um *home address*, mas repassa as conexões para o endereço para um servidor colaborador. O original e o colaborador “parecem” um único servidor.

Diferentes organizações contribuem com máquinas, que serão **compartilhadas** em vários experimentos.

**Problema:**

É preciso garantir que as diferentes aplicações distribuídas não atrapalhem umas às outras. Solução: **virtualização**.

## EXEMPLO: PLANETLAB



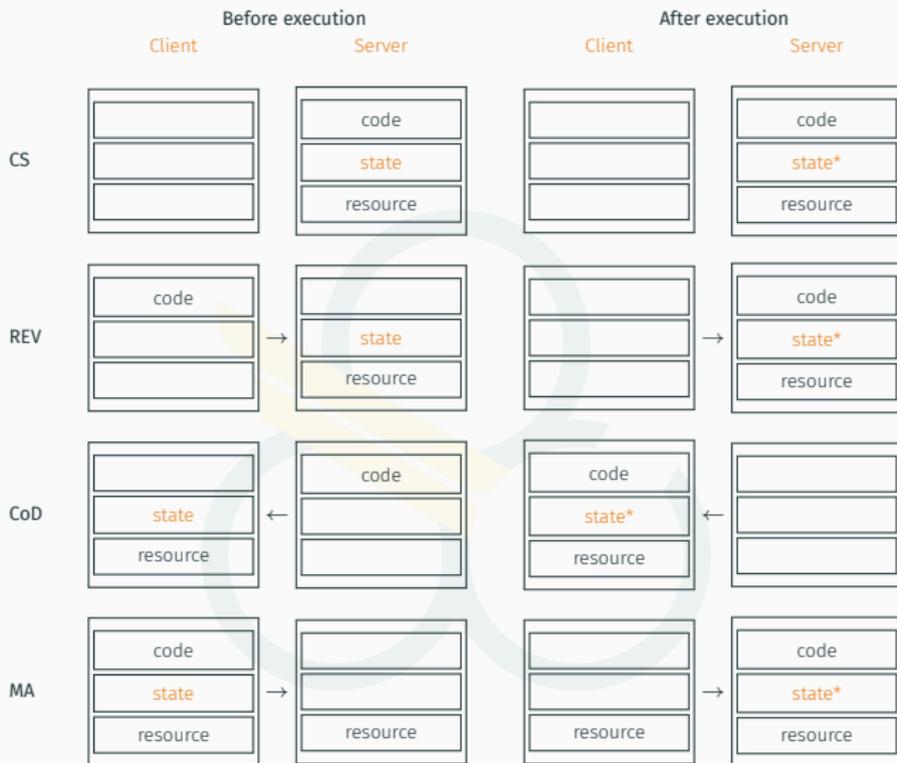
**Vserver:** ambiente independente e protegido com suas próprias bibliotecas, versões do servidor, etc. Aplicações distribuídas são atribuídas a uma **coleção** de Vservers **distribuídas entre múltiplas máquinas físicas** (*slice*).

# MIGRAÇÃO DE CÓDIGO

---

- Abordagens para realização de migração de código
- Migração e recursos locais
- Migração em sistemas heterogêneos

# MIGRAÇÃO DE CÓDIGO: CONTEXTO



CS: Cliente-Servidor  
CoD: Code-on-demand

REV: Remote evaluation  
MA: Agentes móveis

### Componentes do objeto:

**Segmento de código** contém o código real

**Segmento de dados** contém o estado

**Estado da execução** contém o contexto das threads executando o código do objeto

## Mobilidade fraca

Apenas os segmentos de código e dados são migrados (e a execução é reiniciada):

- Relativamente simples, especialmente se o código é portátil
- Duas modalidades: **envio de código** (*push*) e **busca de código** (*pull*)

## Mobilidade forte

Move o componente inteiro, incluindo o seu estado de execução.

- **Migração**: move o objeto inteiro de uma máquina para outra
- **Clonagem**: inicia um clone e o configura para o mesmo estado de execução.

## Problema:

Um objeto usa recursos locais que podem não estar disponíveis no novo local.

## Tipos de recursos

**Fixos:** o recurso não pode ser migrado (ex: hardware)

**Anexado:** a princípio o recurso pode ser migrado, mas migração terá alto custo (ex: banco de dados local)

**Desanexado:** o recurso pode ser facilmente movido junto com o objeto (ex: um cache)

### Ligação objeto–recurso

**Por identificador:** o objeto requer uma instância específica de um recurso (ex: um banco de dados específico)

**Por valor:** o objeto requer o valor de um recurso (ex: o conjunto de entradas no cache)

**Por tipo:** o objeto requer que um determinado tipo de recurso esteja disponível (ex: um monitor colorido)

## GERENCIAMENTO DE RECURSOS LOCAIS

	Desanexado	Anexado	Fixo
<b>ID</b>	MV (ou GR)	GR (ou MV)	GR
<b>Valor</b>	CP (ou MV,GR)	GR (ou CP)	GR
<b>Tipo</b>	RB (ou MV, GR)	RB (ou GR, CP)	RB (ou GR)

*GR = Estabelecer referência global no sistema*

*MV = Mover o recurso*

*CP = Copiar o valor do recurso*

*RB = Religa a um recurso local disponível*

## Problema principal

- A máquina destino pode não ser adequada para executar o código migrado
- A definição de contexto de thread/processo/processador é altamente dependente do hardware, sistema operacional e bibliotecas locais

## Única solução

Usar alguma máquina abstrata que é implementada nas diferentes plataformas:

- Linguagens interpretadas, que possuem suas próximas MVs
- Uma MV Virtual (como vimos no início da aula)

## Migração de imagens: três alternativas

1. Enviar as páginas de memória para a nova máquina e reenviar aquelas que forem modificadas durante o processo de migração
2. Interromper a máquina virtual, migrar a memória, e iniciar a nova máquina virtual
3. Fazer com que a nova máquina virtual recupere as páginas de memória conforme for necessário: processos são iniciados na nova máquina imediatamente e copiam as páginas de memória sob demanda

# DESEMPENHO DA MIGRAÇÃO DE MÁQUINAS VIRTUAIS

## Problema

Uma migração completa pode levar dezenas de segundos. Além disso, é preciso ficar atento ao fato de que um serviço poderá ficar indisponível por vários segundos durante a migração.

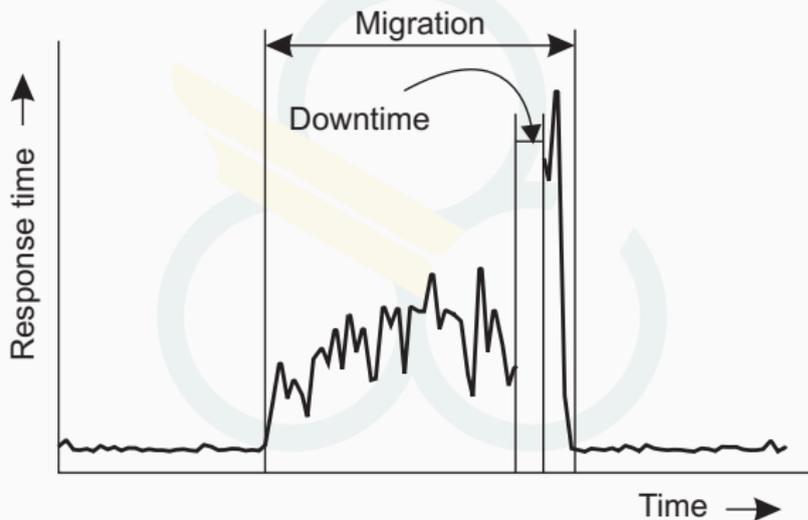


Figura: Medições do tempo de resposta de uma VM durante uma migração