Sistemas Distribuídos
Aula 3

Aula passada

- Processos
- IPC
- Características
- Ex. sinais, *pipes*, *sockets*

Aula de hoje

- Threads
- Kernel level
- User level
- Escalonamento

Motivação: Servidor Web

- Considere Servidor Web que cria um novo processo para atender cada cliente

  - **Muito ineficiente! Por que?**

- Processos fazem parte da mesma computação, compartilham muitas coisas (código, etc)
- Uso de Espaço: PCB, *page tables*, etc
- Uso de Tempo: criar e manter estrutura de dados, troca de contexto, etc.

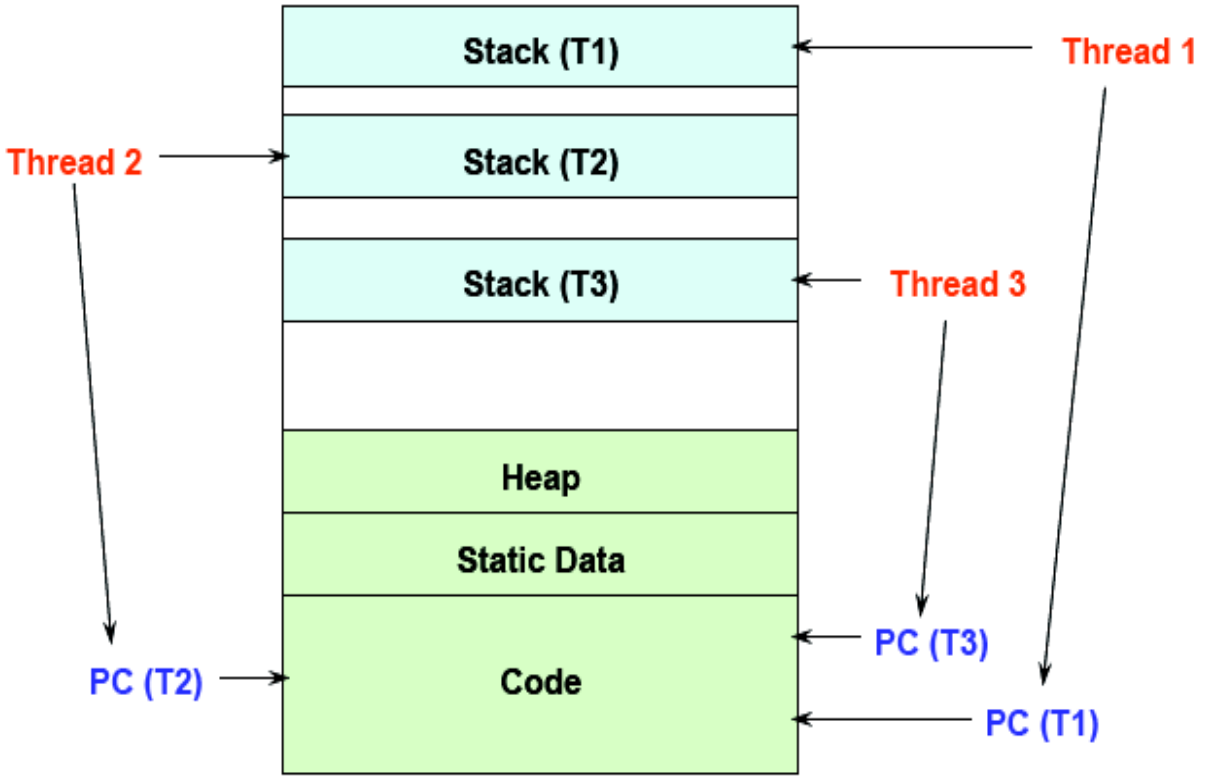
Solução?

Threads

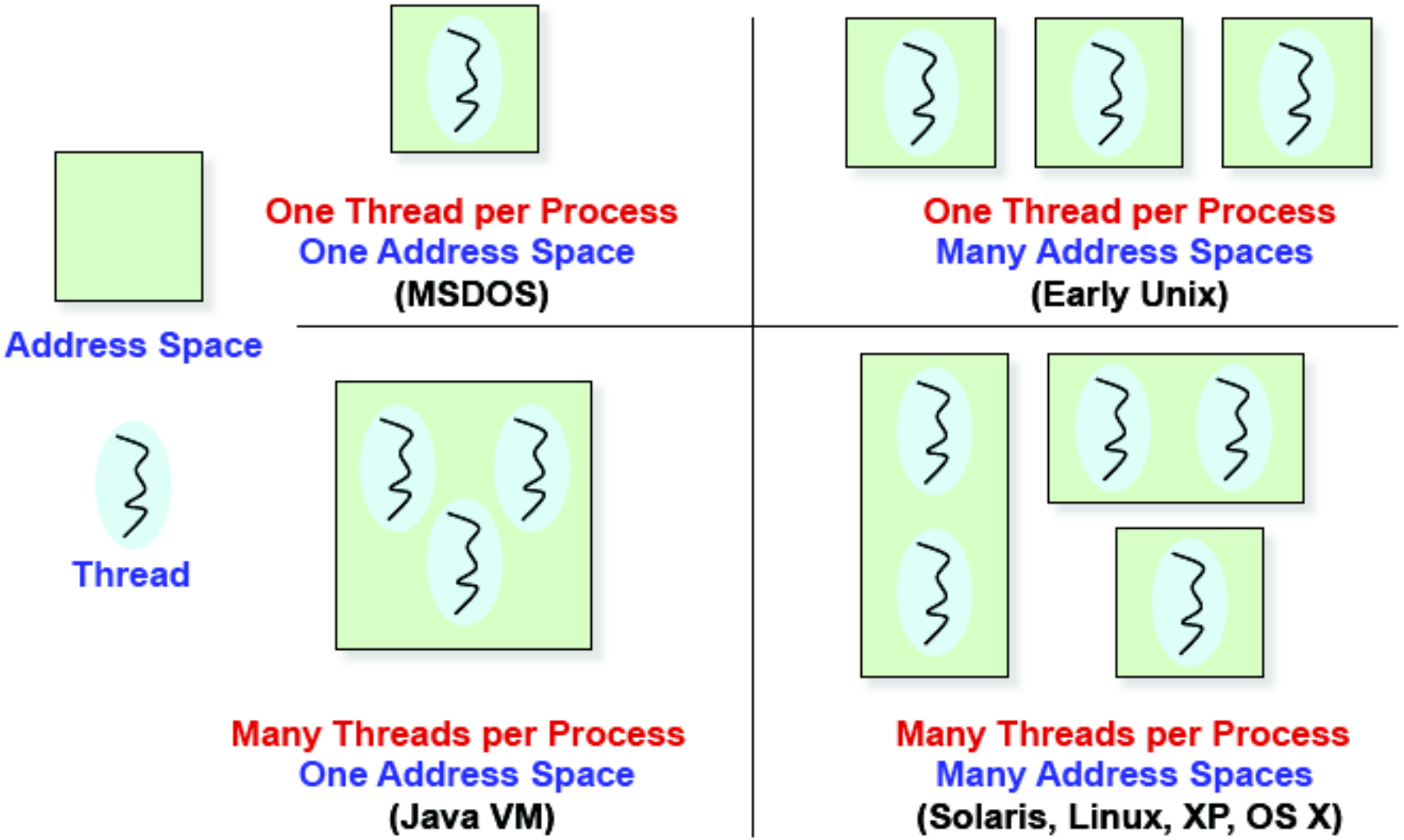
- **Ideia**: separar conceito de processo do estado de execução
  - permitir processos a executarem em mais de um lugar
  - múltiplas linhas de execução: *threads*!
- Estado de uma linha de execução
  - PC, SP, registradores
  - muito mais leve que estado de um processo
- Separação entre processo e thread
  - todo processo tem ao menos uma thread
  - toda thread pertence a exatamente um processo

Threads

- Passa a ser a unidade de escalonamento
  - dentro do espaço de endereçamento do processo
- Processo é estático (estado fixo),
  thread é dinâmica (estado varia com o tempo)
- Cada CPU (core) executa uma thread por vez



Projetando Threads



- *address space*: espaço de endereçamento de um processo (código + dados)

Por que usar *threads*?

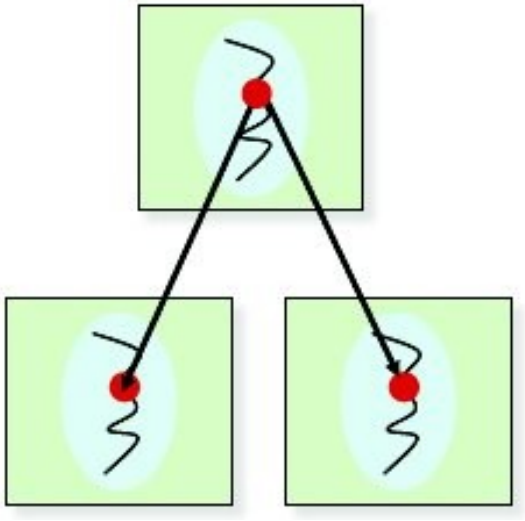
- Facilidade e eficiência do ambiente compartilhado para cooperação
  - programação paralela
- *Multithreading* oferece novas funcionalidades
  - melhor estrutura do programa
  - manuseio de atividades concorrentes
  - aproveitamento de múltiplas CPUs (cores)
  - sobreposição entre I/O e computação

Aumento da complexidade do programa

- ex. uso de sincronização (que veremos)

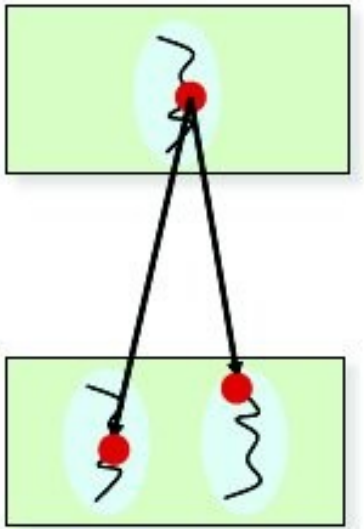
Exemplo do Servidor Web

- Criando novo processo para cada cliente



- Canhão para uma tarefa pequena

- Criando nova *thread* para cada cliente



- Muito mais eficiente (e elegante)

Escalonando Threads

- Como lidar e escalonar threads?
  - antes cada processo tinha uma thread, e agora?
- Duas opções
  - *threads* são explicitamente reconhecidas pelo SO
  - *threads* são escondidas do SO e gerenciadas pelo programa (nível de usuário)
- Aspectos importantes
  - custo de troca de contexto, custo de representar a thread (memória)
  - qual thread do processo deve ser escalonadas?
  - como lidar com chamadas bloqueantes?

Kernel Level Threads

- SO gerencia threads e processos
  - operação de threads implementadas no kernel
  - aka. Lightweight processes (LWP)
- Unidade de escalonamento é threads
  - “processo” (PCB) não é mais escalonado
  - se uma thread fica bloqueada outra do mesmo processo pode executar
- SO decide escalonamento de threads intra e inter processos
  - Justiça?

Vantagens e Desvantagens

- Kernel level threads é mais leve que processos
  - bem menos estado mantido pelo Kernel
- Mas ainda possui ineficiências para "concorrência fina"
  - demandam *system calls* (para criar)
  - precisam ser gerais para acomodar diferentes demandas
- Como ser ainda mais leve?
  - tão leve quanto uma chamada de função?

User Level Threads

- Implementação de threads no nível do usuário
  - gerenciadas por uma biblioteca de execução de threads acoplada ao programa
- Invisíveis para o SO (Kernel)
  - representadas apenas dentro do processo
  - estado de cada thread mantida pelo processo, pela biblioteca de execução de threads
- Criação, troca de contexto, sincronização entre threads feito por chamadas de função dentro do programa!
- Gerenciamento 100x mais rápido que kernel level threads

Limitações

- Limitação de user level threads?

- Por serem invisíveis para o SO, problemas surgem
  - Escalonar um processo que não tem threads prontas para executar
  - Suspender um processo onde uma thread faz uma chamada bloqueante, tendo outras threads prontas
  - Mutiplexar CPU entre processos com número diferente de threads

Kernel e User Level Threads

- **Kernel Level Threads**
  - Integradas com o SO
  - Lentas para criar, manipular, sincronizar
- Disponível em praticamente todos SO
  - Windows, Linux, Mac OS X, Solaris

- **User Level Threads**
  - Gerenciamento feito por biblioteca acoplada ao programa
  - Rápidas para criar, manipular, sincronizar
  - Não reconhecidas pelo SO
- Bibliotecas
  - Pthreads, Windows threads, Java threads

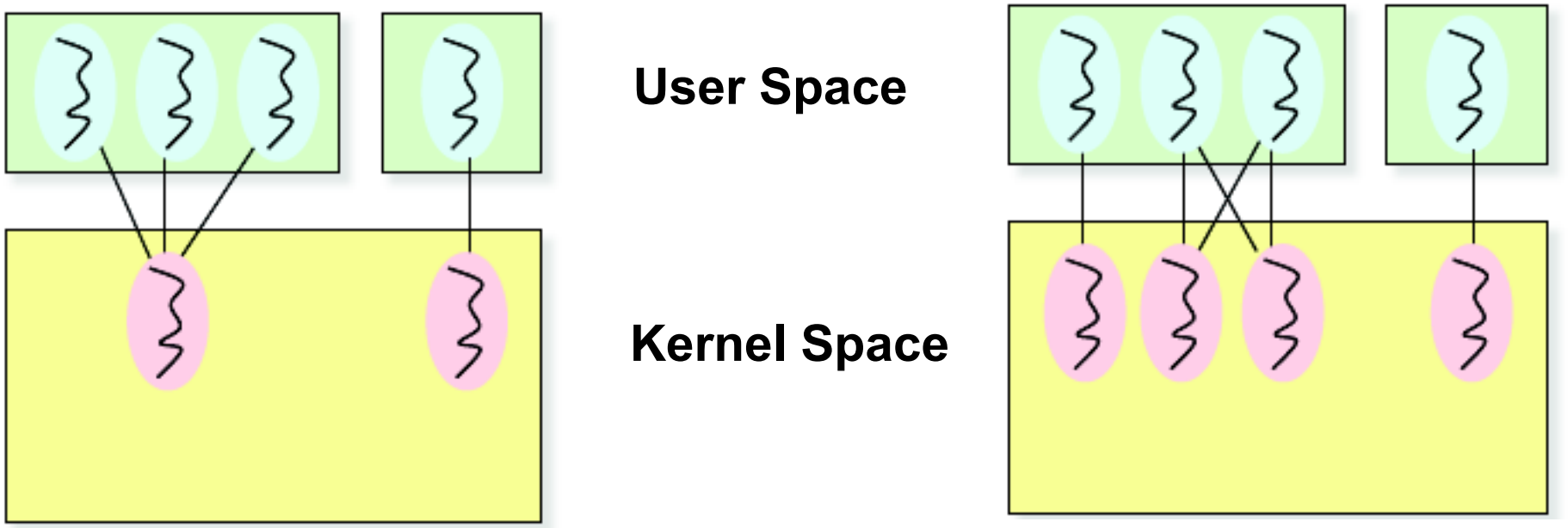
Kernel e User Level Threads

- Como obter o melhor dos dois lados?

Juntando as duas ideias!

- Usar os dois tipos de threads
  - associar ou multiplexar kernel level thread a user level threads
  - biblioteca (user level) conversa com o SO (kernel level)
- Ex. Java Virtual Machine (JVM) é um processo
  - com suporte a java threads (user-level)
- Associar ao JVM mais de uma kernel level thread

Multiplexando Kernel e User Level Threads



- Multiplexar user level threads em uma única kernel level thread por processo
- Multiplexar user level threads em múltiplas kernel level threads por processo

Implementação de Threads

- Muitos aspectos precisam ser resolvidos
  - interface com programa (criar, esperar, etc)
  - troca de contexto
  - escalonamento
  - sincronização
  - sinais
- Como são resolvidos faz parte da especificação da biblioteca de threads
  - diferente entre uma biblioteca e outra!

Cuidado em saber como funciona!

Escalonamento de User Level Threads

- Determina qual thread vai ser colocada em execução
- Usa fila de threads que estão prontas para executar
  - como o escalonador do SO, mas sendo feita pela biblioteca no nível do usuário

- Como trocar de thread (*user level*)?
  - thread em execução está na CPU!

Escalonamento de User Level Threads

- **Não preemptivo**: thread explicitamente libera a CPU, devolvendo o controle à biblioteca
  - thread que roda para sempre trava as outras!
- **Preemptivo**: thread interrompida durante sua excução, assincronamente, devolvendo controle à biblioteca
- Como fazer isto?
  - Sinais. Enviar sinal de alarme ao processo, signal handler pertence a biblioteca

Exemplo

- Um processo, duas user level threads
- Escalonamento não-preemptivo
  - *thread_yield*() força a troca de contexto

Thread Ping

```
while (true) {
    print("ping");
    thread_yield();
}
```

Thread Pong

```
while (true) {
    print("pong");
    thread_yield();
}
```

- Escalonamento preemptivo
  - thread interrompidas por *timer* (ex. 1ms)

Thread Ping

```
while (true) {
    print("ping");
}
```

Thread Pong

```
while (true) {
    print("pong");
}
```

- O que será impresso em cada caso?