Prof. Ricardo Inácio Álvares e Silva

Threads

Sistemas Operacionais - Introdução a Processos

Escopo da aula

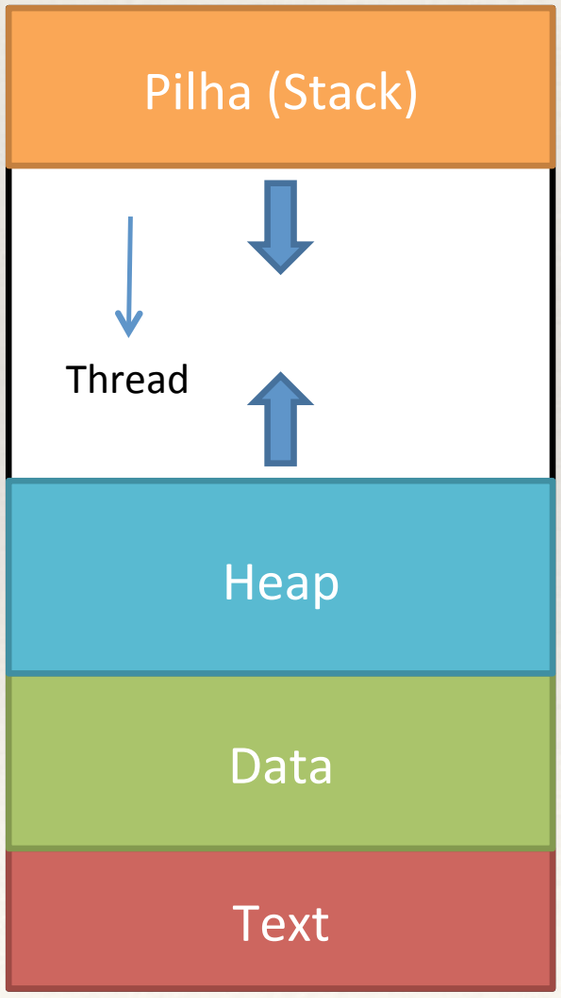
- Threads
  - Definição
  - Motivação
  - Exemplo
  - Atributos
  - Modelos de threads em S.O.
  - Bibliotecas
  - Banco de Threads

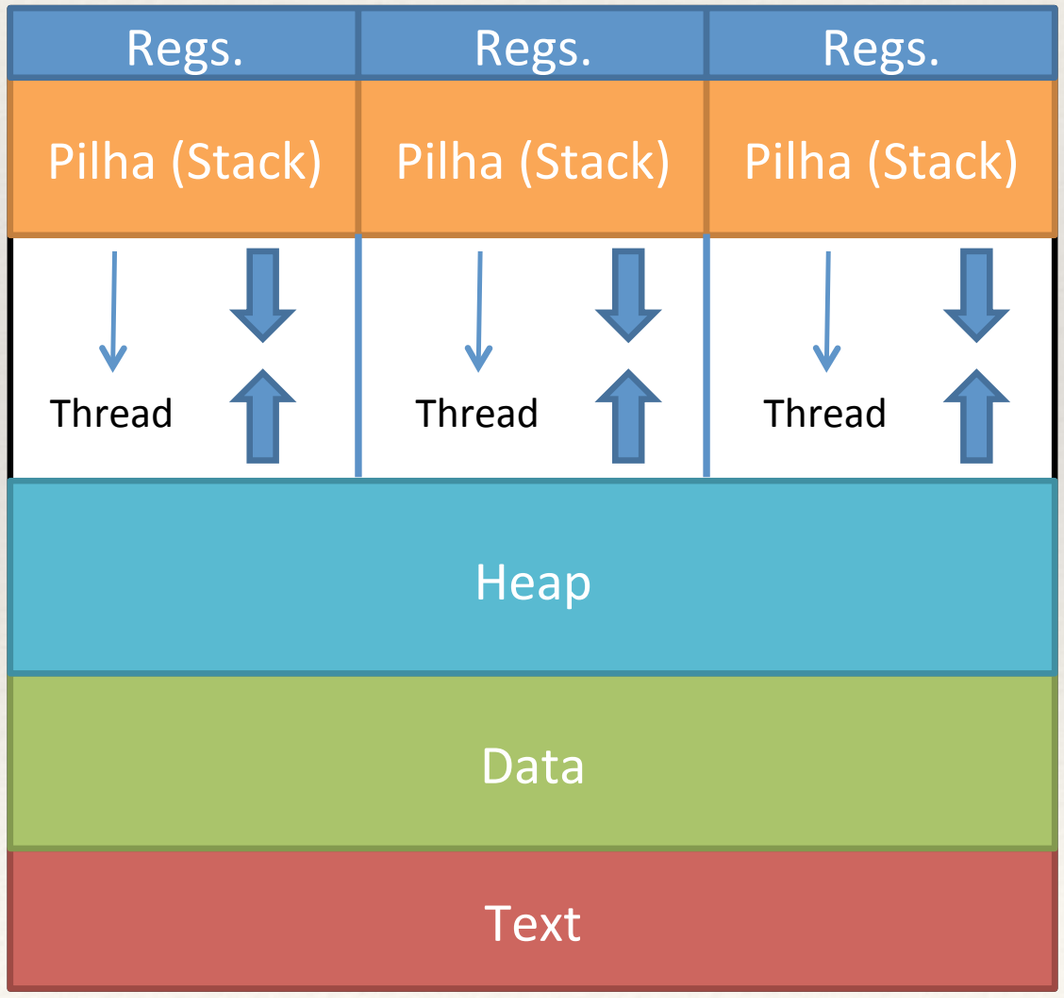
Para relembrar…

- **Processo**: uma instância de um programa em execução, e todos os atributos necessários ao controle dessa execução.
  - Cada processo possui uma thread.
- **Thread**: linha de execução seqüencial de instruções do programa executado por um processo.
- **Seqüencial** significa que executa apenas uma instrução por vez, e sempre em ordem.

Múltiplas threads em um único Processo

- Em muitos sistemas operacionais, é possível um processo ter múltiplas threads





Motivação

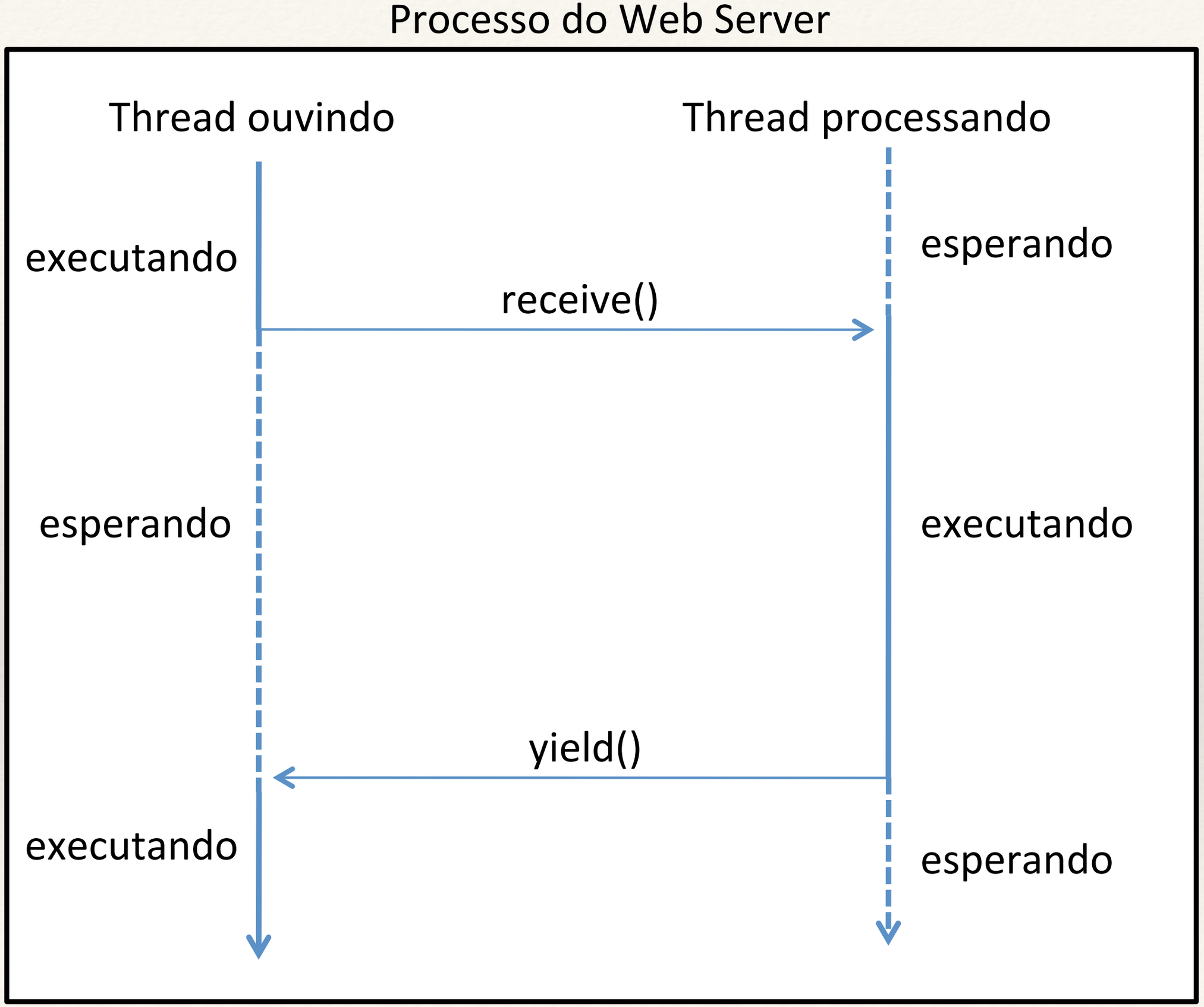
- Qual a utilidade de se permitir múltiplas *threads* em um processo?
  - *Um programa pode utilizar várias linhas de execução colaborativas*
- Não seria melhor lançar múltiplos processos para realizar tarefas combinadas?
  - *Para muitas situações, não!*
- *Threads* de um mesmo processo têm **acesso a um espaço de memória comum**
  - Podem compartilhar trabalho facilmente e a custo baixo de desempenho
- A criação de *threads* em um processo é mais eficiente que a criação de novos processos, que envolve inicializar um novo PCB e inicializar um espaço de memória (duplicar, no caso do fork())

Motivação

- A criação de *threads* em um processo é mais eficiente que a criação de novos processos
  - Um novo processo que envolve inicializar um novo PCB e inicializar um espaço de memória
- O escalonamento de *threads* é mais ágil que de processos inteiros
  - Tipicamente, de 10 a 100 vezes mais rápido
- Um processo que faz uma chamada ao sistema que o coloca em estado de espera pode fazer **sobreposição de threads**
  - Uma *thread* entra em espera, enquanto as outras continuam prontas ou em execução
- Programas executando em computadores com múltiplos processadores podem explorar o **paralelismo real**

Um exemplo...

- Um servidor web necessita se comunicar com os clientes pelo dispositivo de rede
- A chamada ao sistema `receive()` é utilizada para receber a requisição do cliente
- Ao realizar um `receive()`, o processo do servidor web fica parado esperando uma mensagem ser recebida
- Porém, o servidor web possui várias outras funções, como gerar as páginas que serão pedidas por este e outros clientes
- O ideal é o servidor web possuir dois processos:
  - um processo ficar ouvindo clientes
  - enquanto outro processa as páginas
- Devido ao alto custo de comunicação entre processos, **é inviável**
  - a solução é a sobreposição de threads


Processo do Web Server
Thread ouvindo
Thread processando
executando
esperando
receive()
esperando
executando
yield()
executando
esperando

Atributos de *threads*

- Assim como processos, *threads* fazem troca de contextos, apesar de mais simples
- Para ser possível a troca de contexto, é necessário guardar atributos de execução de *threads*
- Um processo pode ter múltiplas *threads*
  - todas as *threads* de um processo dividem os atributos do processo
  - cada *thread* tem seus próprios **atributos de *thread***

- Os atributos de *threads* ficam na **tabela de *threads***

Por Processo	Por Thread
Espaço de endereçamento	Contador de Programa (PC)
Variáveis globais	Registradores
Arquivos abertos	Pilha (Stack)
Processos filhos	Estado
Alarmes programados	
Sinais	

- Onde fica a tabela de *threads*?
  - Espaço de *kernel* ou usuário?

Modelos de *threads*

- A tabela de *threads* pode estar no espaço
  - de *kernel*
  - de usuários
  - ou ambos!
- A localização da tabela de threads possui implicações para o sistema como um todo
- É o que define o **modelo de *threads*** de um sistema

Modelo "muitos para um"

- A tabela de *threads* fica no espaço de memória de usuário
- Isso significa que o escalonamento de *threads* é feito pelo próprio programa do processo, tipicamente uma biblioteca
- Do ponto de vista do S.O (o *kernel*), o sistema não suporta múltiplas *threads*
  - cada processo simula a existência de *threads*
  - simulação por conta própria, não há um modelo padronizado e geral para o sistema

- Dentre os benefícios dessa abordagem:
  - rápido desempenho em trocas e criação de threads, pois não dependem de chamadas ao sistema e gerenciamento diretamente pelo *kernel*
  - possibilidade de usar *threads* em qualquer sistema
- Desvantagens:
  - uma *thread* bloqueada por uma chamada ao sistema bloqueia todo o processo, impossibilita sobreposição de *threads*
  - problemas para explorar o paralelismo real

Modelo "um para um"

- A tabela de *threads* fica no espaço de memória do *kernel*
- Isso significa que o escalonamento de *threads* é feito pelo S.O, como se fosse um processo normal
- Benefício: é possível a sobreposição de *threads* em um mesmo processo
- Desvantagem:
  - criação e escalonamento de *threads* possui desempenho inferior, pois envolve
    - chamadas ao sistema
    - interrupção
    - trocas de contexto
- É o modelo utilizado por Linux e Windows

Modelo "muitos para muitos"

- Modelo que suporta os dois formatos anteriores
- Possibilita utilização ideal de recursos, desde que se saiba separar o que vai ser feito por cada *thread*
- Complexidade de programação
  - Quais *threads* devem ser do sistema?
  - Quais *threads* devem ser do processo?

Bibliotecas de *threads*

- APIs (*Application Programming Interface*) para criação e gerenciamento de threads
- **Pthreads**
- **Win32**
- **Java**

Pthreads

- Biblioteca POSIX, define cerca de 60 chamadas comuns
- Pode representar qualquer um dos modelos de *threads*
  - depende da implementação específica de cada S.O.
- Implementada pela maioria dos sistemas UNIX

Algumas chamadas ao sistema do Pthreads

Chamadas	Descrição
pthreads_create	Cria uma nova *thread*
pthread_exit	Termina a *thread* que fez a chamada
pthread_join	Fica em espera até que uma *thread* específica termine
pthread_yield	Libera a CPU para uma outra *thread*
pthread_attr_init	Cria e inicializa uma estrutura de atributos para uma *thread*
pthread_attr_destroy	Elimina uma estrutura de atributos de uma *thread*

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

const int NUM_THREADS = 10;

void * ola_thread(void *tid){
    /* Essa função escreve o ID da thread e termina */
    printf("Ola Mundo! Saudacoes da thread %d0\n", *((int*)tid));
    pthread_exit(NULL);
}

int main(int argc, char * argv[]){
    /* O programa principal cria 10 threads e termina */
    pthread_t threads[NUM_THREADS];
    int erro, i;
    for (i = 0; i < NUM_THREADS; i++){
        printf("Main no controle. Criando thread %d0...\n", i);
        erro = pthread_create(&threads[i], NULL,
                        ola_thread, (void *)i);
        if (erro != 0){
            printf("Opa, pthread_create retornou o codigo de erro
%d0\n", erro);
            exit(-1);
        }
    }
    return 0;
}
```

Win32

- Biblioteca de threads do Windows.
- Implementa modelo de threads um para um.

```
#include <windows.h>
/* dado compartilhado entre threads */
DWORD Soma;

DWORD WINAPI Somatorio(LPVOID Param) {
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Soma += i;
    return 0;
}

int main(int argc, char **argv[]) {
    DWORD ThreadId;
    int Param = atoi(argv[1]);
    HANDLE ThreadHandle = CreateThread(
        NULL,          // atributos de segurança padrão
        0,             // tamanho da pilha padrão
        Somatorio,     //função executada pela nova thread
        &Param,        // parâmetro passado para a função
        0,             // flags de criação padrão
        &ThreadId);    // ID da nova thread

    WaitForSingleObject(ThreadHandle,INFINITE);
    CloseHandle(ThreadHandle);
    printf("soma = %d\n",Soma);
}
```

Java

- Possui uma API padrão para manipular threads em programas que rodam sobre a JVM
  - Na realidade, essas chamadas utilizam a biblioteca padrão do S.O em que a JVM está rodando
- O modelo de threads depende do S.O
  - um mesmo aplicativo Java pode variar no funcionamento de acordo com o S.O utilizado
- Qualquer classe que implemente a interface `Runnable` pode ter o seu método `run()` invocado para iniciar uma nova *thread*.

```
class Instrumento implements Runnable {
    public Intrumento(String som) {
        this.som = som;
    }

    public void run() {
        System.out.println(som);
    }

    private String som;
}


public class Maestro {
    public static void main(String... args) {
        String[] orquestra = {"violino","trombone","baixo"};
        for (int i = 0; i < orquestra.length; i++) {
            Thread thrd = new Thread(
                    new Instrumento(orquestra[i]));
            thrd.start();
        }

        System.out.println("Parou!");
    }
}
```