# SSC150 – Sistemas Computacionais Distribuídos

**Processos em Sistemas Distribuídos**

7ª aula
22/04/10

Profa. Sarita Mazzini Bruschi
sarita@icmc.usp.br

Slides baseados no material de:
Prof. Rodrigo Mello (USP / ICMC)
Prof. Edmilson Marmo Moreira (UNIFEI / IESTI)

# Processos

- Sistemas Operacionais mais antigos:
  - Cada processo tem um único espaço de endereçamento e um único fluxo de controle
- Existem situações onde é desejável ter múltiplos fluxos de controle compartilhando o mesmo espaço de endereçamento:
  - Solução: threads

# Threads (processos leves)

- Execução estritamente sequencial, tendo seu próprio contador de programa e pilha
- As threads partilham a CPU do mesmo modo que os processos
  - Em um sistema multiprocessado, elas podem executar em paralelo
- As threads podem criar threads filhas e ficam bloqueadas como os processos
- Todas as threads têm o mesmo espaço de endereçamento, portanto possuem as mesmas variáveis globais
- Uma thread pode ler e escrever na pilha de outra thread

# Processos e Threads

- Conceitos de threads e processos

| **Threads** |
| --- |
| Contador de Programa |
| Pilha |
| Registradores |
| Threads filhas |
| Estado |

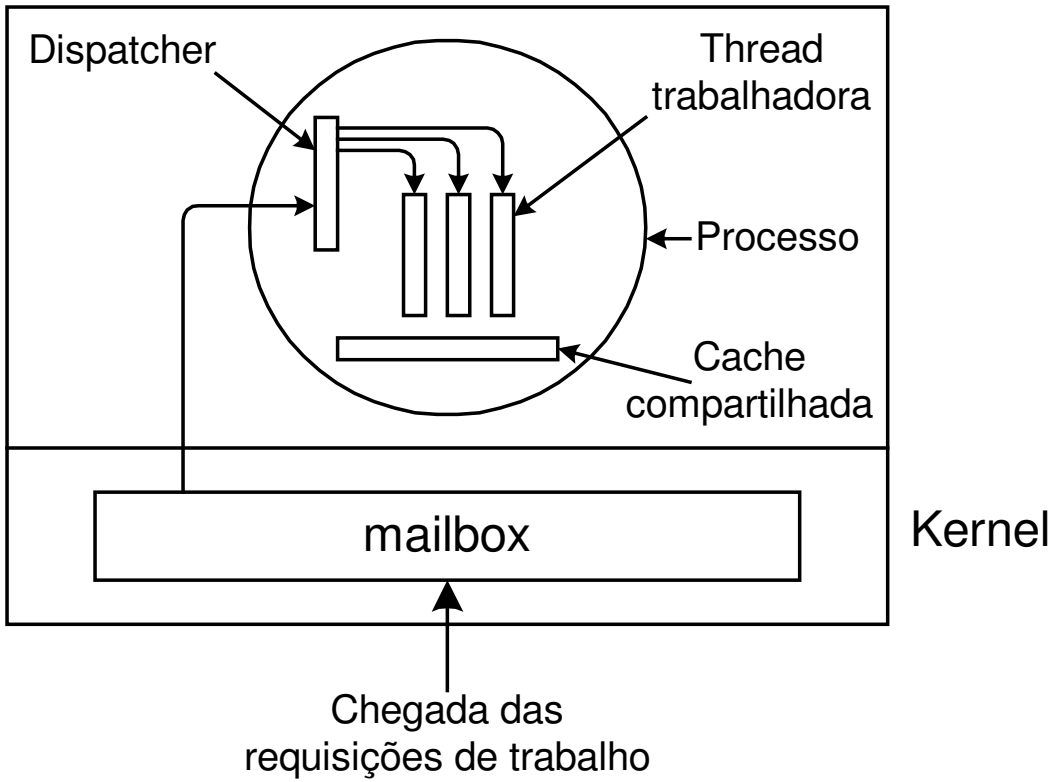
| **Processos** |
| --- |
| Espaço de Endereçamento |
| Variáveis globais |
| Arquivos abertos |
| Processos filhos |
| Temporizadores |
| Sinais |
| Semáforos |

- Assim como os processos, as threads podem estar nos estados: executando, bloqueado, pronto ou finalizado

# Uso de threads

- As threads foram criadas para permitir paralelismo combinado com execução sequencial e chamadas do sistema bloqueantes
- Possíveis organizações de um sistema multithread:
  - Dispatcher/worker (distribuidor/trabalhador)
  - Team (time)
  - Pipeline
- Exemplo: servidor de arquivos

# Uso de threads
# Modelo Distribuidor/Trabalhador



# Uso de threads
# Modelo Distribuidor/Trabalhador

- A thread Distribuidora lê requisições de um sistema de mailbox
- Após examinar a requisição, ela escolhe uma thread Trabalhadora desocupada, e passa a requisição para essa thread
- A thread Distribuidora “acorda” a thread trabalhadora, que pode estar bloqueada por um semáforo, por exemplo
- Quando a trabalhadora ficar bloqueada, outra thread assumirá o estado de executando

# O mesmo problema resolvido sem thread

- Operando com um sistema tradicional:
  - O servidor de arquivos pega uma requisição
  - Executa até que ela seja completada antes de ir para a próxima
  - Enquanto espera pelo disco, o servidor espera desocupado e não processa nenhuma outra requisição

# O mesmo problema resolvido sem thread

- Executando o servidor como uma máquina de estado finito:
  - Quando uma requisição chega, um processo examina
  - Se ela não pode ser satisfeita através da cache, uma mensagem é enviada para o disco
  - Entretanto, ao invés de bloquear, ele registra o estado corrente da requisição em uma tabela e fica livre para tratar a próxima requisição
  - Se a próxima for uma resposta do disco, as informações da tabela devem ser usadas para restaurar o estado de computação e a resposta processada
- Esse modelo nada mais é do que uma simulação das threads

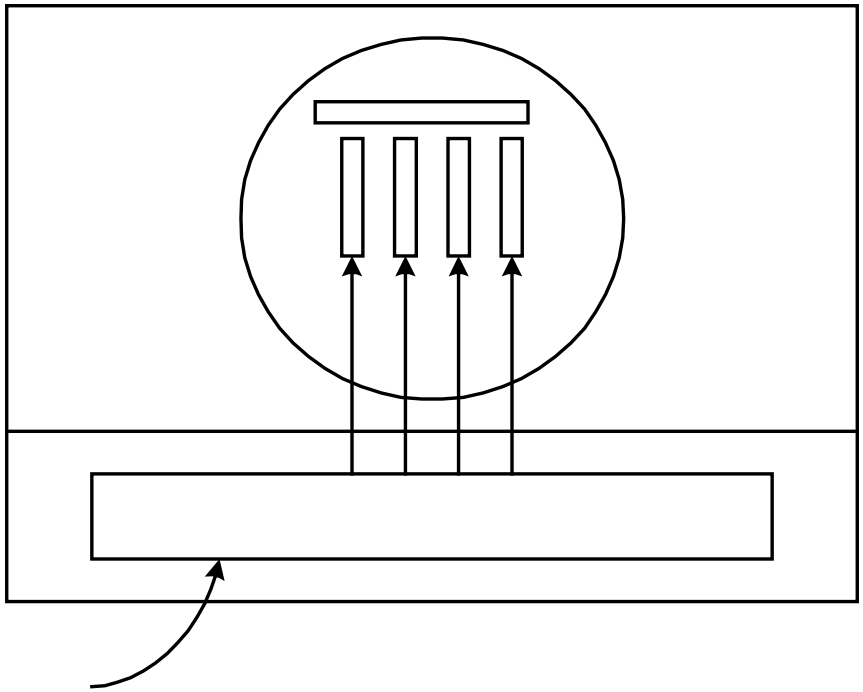
# Resumo

- Threads torna possível manter a idéia de processo sequencial que usa chamadas bloqueantes e, ao mesmo tempo, consegue paralelismo
- Chamadas bloqueantes tornam a programação mais fácil e o paralelismo aumenta o desempenho

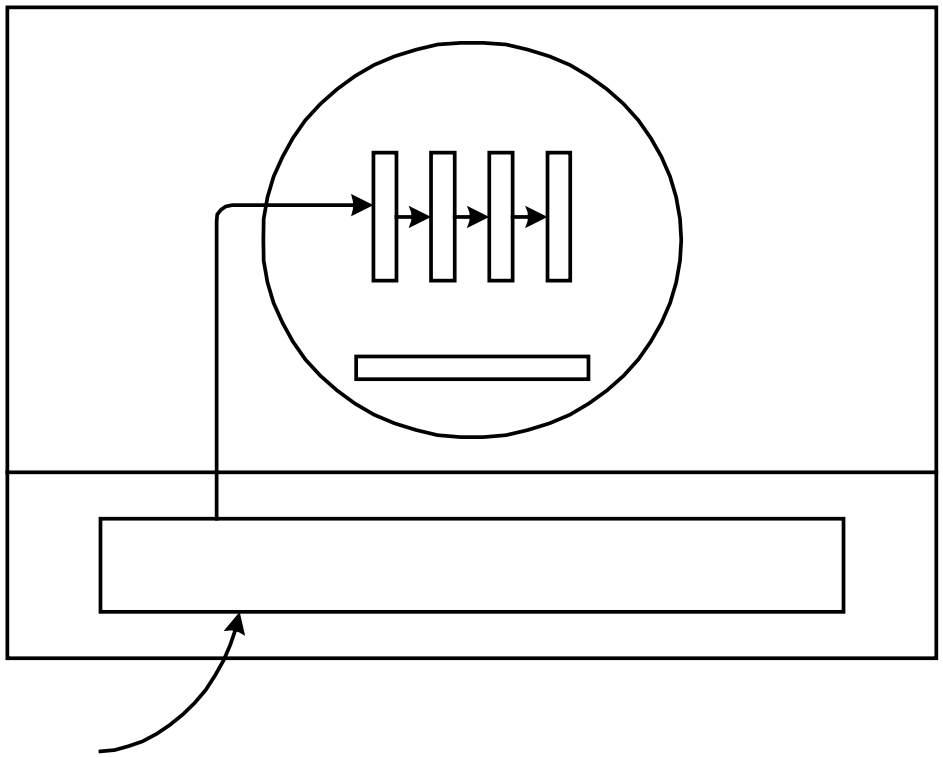
# Resumo

| Modelo | Características |
|---|---|
| Threads | Paralelismo, chamadas bloqueantes |
| Processo | Sem paralelismo, chamadas bloqueantes |
| Máquina de estado finito | Paralelismo, chamadas não bloqueantes |

# Uso de threads
# Modelo Time



# Uso de threads
# Modelo Pipeline



- A princípio todas as threads possuem a mesma função

# Uso de threads
# Modelo Pipeline

- Cada thread é responsável por uma parte do trabalho, sendo que o resultado produzido por uma thread é passado como entrada da próxima thread
- Utilizado principalmente quando se usa o pipe do unix/linux
  - ps –usarita | grep emacs
  - history | grep vim

# Threads em Java
# Primeira maneira de criação

- Criação da thread:
  - Definir uma nova classe derivada da classe Thread
  - Redefinir o método run()
- A definição dessa nova classe não cria a nova thread
  - A criação é feita através do método start()
    - Aloca memória e inicializa uma nova thread na JVM
    - Chama o método run(), tornando a thread elegível para ser executada pela JVM

# Threads em Java
# Exemplo

```
class Worker1 extends Thread
{
    public void run() {
        System.out.println("Eu sou uma thread criada");
    }
}

public class First
{
    public static void main(String args[]) {
        Thread runner = new Worker1();
        runner.start();
        System.out.println("Eu sou a thread principal");
    }
}
```

# Threads em Java
# Segunda maneira de criação

- Criação da thread
  - Definição de uma classe que implemente a interface Runnable.

```
public interface Runnable
{
    public abstract void run();
}
```

  - Definição de um método run()
- A implementação da classe Runnable é semelhante à extensão, exceto que “extends Thread” é substituído por “implements Runnable”

# Threads em Java
## Segunda maneira de criação

- Como a nova classe não estende Threads, ela não tem acesso aos métodos estáticos ou de instância, como por exemplo, o método start(), da classe Thread
- Porém, um método start() ainda é necessário, pois é ele que cria uma nova thread de controle
- Um novo objeto Thread deve ser criado, recebendo um objeto Runnable em seu construtor
- Quando a thread é criada com o método start(), a nova thread inicia a execução no método run() do objeto Runnable.

# Threads em Java
# Exemplo

```
class Worker2 implements Runnable
{
    public void run() {
        System.out.println("Eu sou uma thread criada");
    }
}

public class Second
{
    public static void main(String args[]) {
        Thread thrd = new Thread(new Worker2());
        thrd.start();
        System.out.println("Eu sou a thread principal");
    }
}
```

# Multitasking

- Duas maneiras de se tratar multitarefa:
  - Thread-per-client (Servidores concorrentes)
    - Uma nova thread é instanciada para cada nova conexão de cliente
  - Thread pool
    - Um conjunto fixo de threads previamente instanciadas trabalha para manipular as conexões dos clientes

# Multitasking e Sockets

# Servidor

```
import java.net.*;
import java.io.*;
public class Server extends Thread {
        private ServerSocket server;
        public Server(int port) throws Exception {
                server = new ServerSocket(port);
        }
        public Socket accept() throws Exception {
                return server.accept();
        }
        public void run() {
                try {
                    while (true) {
                            (new RequestClient(accept())).start();
                    }
                } catch (Exception e) {}
        }
        public static void main(String args[]) throws Exception {
                Server s = new Server(1234);
                System.out.println("Iniciando Servidor...");
                s.start();
        }
}
```

Classe Server derivada da classe Thread

Redefinição do método run()

# Multitasking e Sockets
# RequestClient

```
import java.net.*;
import java.io.*;
public class RequestClient extends Thread {
        private Socket client;
        private PrintStream ps;
        private BufferedReader br;
        public RequestClient(Socket client) throws Exception {
                this.client = client;
                mountingChannels(this.client);
        }
private void mountingChannels(Socket socket) throws Exception {
        ps = new PrintStream(socket.getOutputStream());
        br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        }
public void writeln(String str) throws Exception {
        ps.println(str);
        }
public String readln() throws Exception {
        return br.readLine();
        }
public void run() {
        try {
                /*Tratando o request do cliente*/
                String str = readln();
                try {
                  FileInputStream f = new FileInputStream(str);
                  BufferedReader line = new BufferedReader(new InputStreamReader(f));
                  String buffer = "";
                  String all = "";
                  do {
                                buffer = line.readLine();
                                all += buffer;
                  } while (buffer != null);
                  writeln(all);
                } catch (Exception err) { writeln("Error or cannot find file..."); }
        } catch (Exception e) {}
        }
}
```

# Multitasking e Sockets

## Client

```
import java.net.*;
import java.io.*;

public class Client {
	private Socket client;
	private PrintStream ps;
	private BufferedReader br;

	public Client(String serverName, int port) throws Exception {
		client = new Socket(serverName, port);
		mountingChannels(client);
	}

	private void mountingChannels(Socket socket) throws Exception {
		ps = new PrintStream(socket.getOutputStream());
		br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
	}

	public void writeln(String str) throws Exception {
		ps.println(str);
	}

	public String readln() throws Exception {
		return br.readLine();
	}

	public static void main(String args[]) throws Exception {

		if (args.length < 3) {
			System.out.println("usage: java Client serverName portNumber fileName");
			System.exit(-1);
		}

		/*conectando-se ao servidor*/
		Client c = new Client(args[0], (new Integer(args[1])).intValue());

		c.writeln(args[2]);
		System.out.println(c.readln());
	}
}
```