

SISTEMAS DISTRIBUÍDOS

INF2545



Sistemas Distribuídos

- o que são: "coleção de máquinas independentes que aparecem para o usuário como um único sistema coerente"
 - que tipo de usuário?
 - » programador é usuário?
 - o que é "coerente"?
 - » o conceito de transparência
- um sistema distribuído é uma coleção de máquinas independentes que são usadas em conjunto para executar uma tarefa ou prover um serviço.



para que queremos SDs?

- custo e desempenho
 - paralelismo e computação distribuída
- escalabilidade
 - facilidade de aumentar recursos
- distribuição inerente
 - dispersão geográfica e social
 - compartilhamento de recursos
- confiabilidade
 - redundância



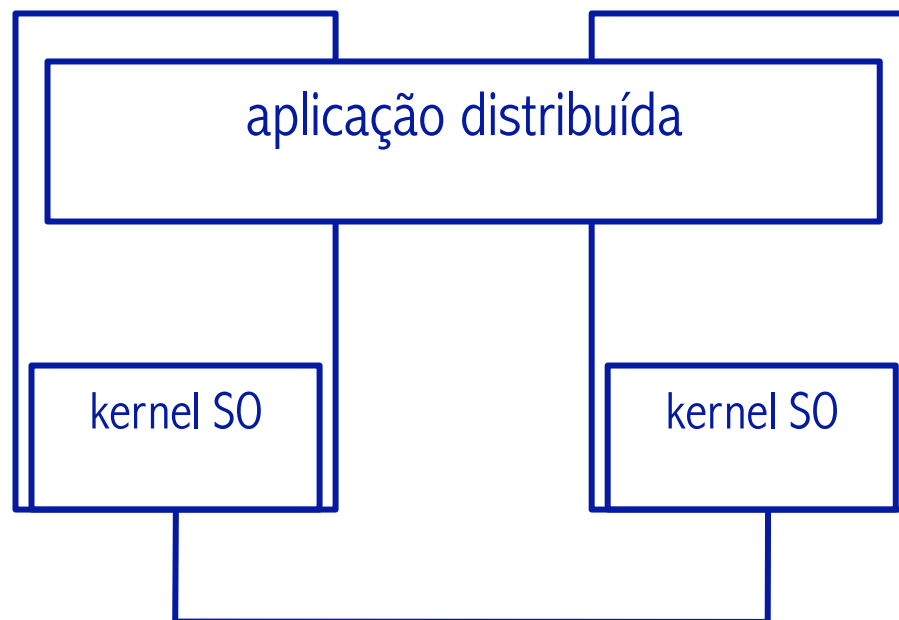
mas...

- rede tem que ser levada em consideração
 - desempenho e falhas
- segurança
 - distribuição introduz problemas inexistentes em sistemas centralizados
- complexidade de software



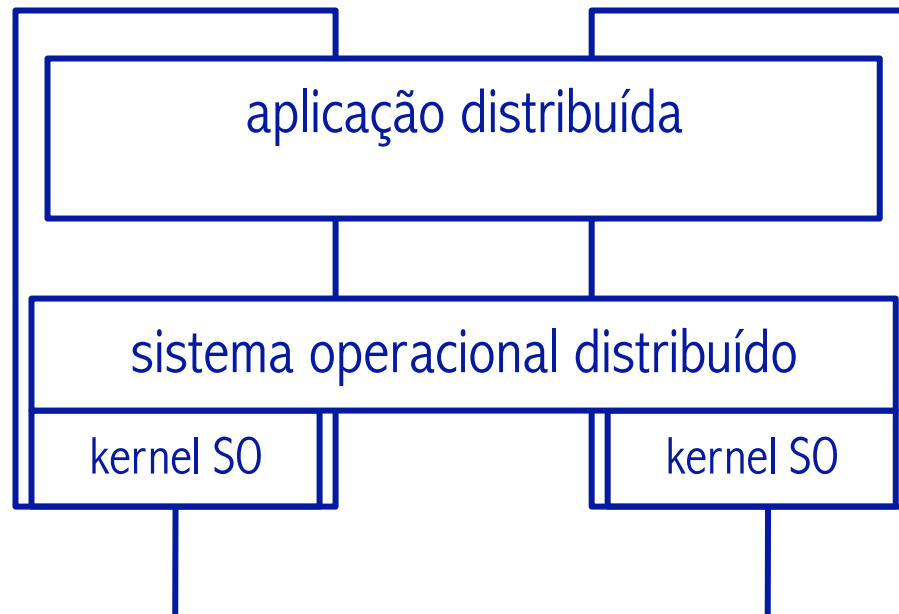
arquiteturas de interesse

- multicomputadores: cada um com sua memória e processador, interligados por redes
 - construção de aplicação distribuída sobre recursos de redes de SO pode ser árdua



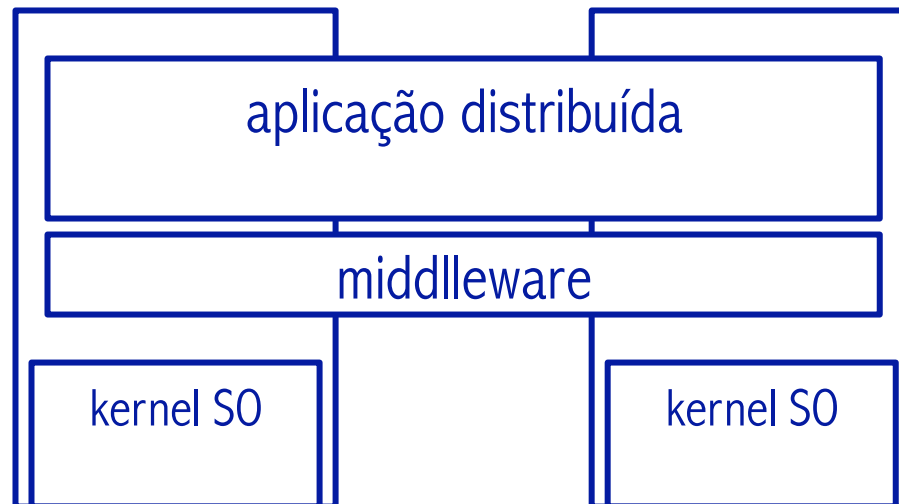
SOs distribuídos

- interesse especial nos anos 80
- sistemas que controlariam o conjunto de recursos de várias máquinas
 - remanescentes importantes:
 - » servidores de arquivos, técnicas de segurança, ...



Middleware

- serviços e abstrações que facilitam o desenvolvimento de aplicações distribuídas



estudo de SDs

- livros clássicos de sistemas distribuídos
 - comunicação entre processos
 - sincronização
 - sistemas de arquivos
 - segurança
 - confiabilidade
- nesse curso, maior ênfase na programação de SDs



programação

- requisitos de diferentes ambientes
 - redes locais e geográficas
 - baixo e alto acoplamento
- facilidades de programação importantes para diferentes classes de aplicações
 - ou mesmo para diferentes interações dentro da mesma aplicação
- necessidades
 - modelos de programação suportados:
 - » cliente-servidor, p2p, computação móvel, ...
 - interesse especial em Lua e contribuição de linguagem interpretada



programa do curso

- Introdução
 - processos e concorrência. threads. eventos.
- Comunicação
 - troca de Mensagens. abstrações. chamada remota de procedimentos e métodos. publish/subscribe. comunicação em grupo.
 - código móvel
- Arquiteturas
 - cliente-servidor
 - p2p
 - agentes móveis
 - eventos...
- Sincronização e Coordenação
 - multicast confiável e ordenado
 - exclusão mútua
- Outros
 - nomes
 - segurança
 - replicação
 - tolerância a falhas



discussão

- facilidade de desenvolvimento
- desempenho
- transparência
- escalabilidade
- flexibilidade



Avaliação

- 4 trabalhos (implementação) - grupo
- resumos e críticas de artigos - individuais
- 1 prova – individual



Bibliografia

- A. Tanenbaum e M. van Steen. Distributed Systems: Principles and Paradigms. Prentice-Hall, 2002.
- G. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley, 2000.
- surveys e artigos
 - alguns “clássicos”:
 - » Andrews, Gregory. Paradigms for Process Interaction in Distributed Programs. ACM Computing Surveys , 23(1), mar 91
 - » Briot, J., Guerraoui, R., and Lohr, K. 1998. Concurrency and distribution in object-oriented programming. ACM Comput. Surv. 30 (3). set 98.
 - outros
- Roberto Ierusalimschy. Programming in Lua. lua.org, 2006.



introdução

- processos e concorrência



Processos e Concorrência

- processo: modelo de execução sequencial
 - "programa em execução"
 - código, dados globais, pilha de execução
 - estrutura presente em qualquer sistema operacional
- concorrência
 - modelo de linhas "simultâneas" de execução
- concorrência e distribuição
 - aplicações distribuídas envolvem a execução concorrente de processos em várias máquinas
 - uso de concorrência local:
 - » atendimento a comunicações concorrentes
 - » sobreposição de comunicação e processamento



alternativas p/ concorrência

1. multithreading

- várias linhas de execução compartilham globais com escalonamento preemptivo
- surgido de estudos de sistemas operacionais
- dificuldades de sincronização
- exemplos: threads em C (posix) e em Java



exemplo em Java

```
public class ThreadsDorminhocas {  
    public static void main(String[] args) {  
        new ThreadDorminhoca("1");  
        new ThreadDorminhoca("2");  
        new ThreadDorminhoca("3");  
        new ThreadDorminhoca("4");  
    }  
}
```



threads em Java

```
class ThreadDorminhoca extends Thread {
    int tempo_de_sono;
    public ThreadDorminhoca(String id) {
        super(id);
        tempo_de_sono = (int) (Math.random() * 5000);
        System.out.println("Tempo de sono da thread "+id+
            ": "+tempo_de_sono+"ms");

        start();
    }
    public void run() {
        try {
            sleep(tempo_de_sono);
        } catch (InterruptedException exception) {
            System.err.println(exception);
        }
        System.out.println("thread "+getName()+" acordou!");
    }
}
```

transferência de controle implícita!
(preempção)



condições de corrida

```
class Conta {
  private int saldo;
  public Conta (int ini) {
    saldo = ini;
  }
  public int veSaldo() {
    return saldo;
  }
  public void deposita(int dep) {
    for (int i=0; i<dep; i++) { // artificial!!
      saldo++;
    }
  }
}
```



condições de corrida

```
public class ThreadsEnxeridas {  
    public static void main(String[] args) {  
        int repet = 20;  
        Conta cc = new Conta(0);  
        (new ThreadEnxerida("1", cc, repet)).start();  
        (new ThreadEnxerida("2", cc, repet)).start();  
        (new ThreadEnxerida("3", cc, repet)).start();  
        (new ThreadEnxerida("4", cc, repet)).start();  
    }  
}
```



condições de corrida

```
class Conta {
    private int saldo;
    public Conta (int ini) {
        saldo = ini;
    }
    public int veSaldo() {
        return saldo;
    }
    synchronized public void deposita(int dep) {
        for (int i=0; i<dep; i++) {
            try {
                Thread.sleep(10); // para escalonador agir!
            }
            catch (InterruptedException exception) {
                System.err.println(exception);
            }
            saldo++;
        }
    }
}
```



alternativas ao modelo multithread clássico

2. modelos orientados a eventos

- cada evento tratado até o final
- programa como máquina de estado

3. multitarefa sem preempção

- co-rotinas em Lua

4. multithreading com troca de mensagens

- Erlang

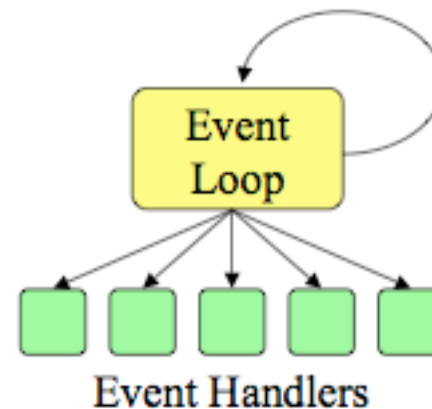


alternativas p/ concorrência

- eventos - descrição Ousterhout:

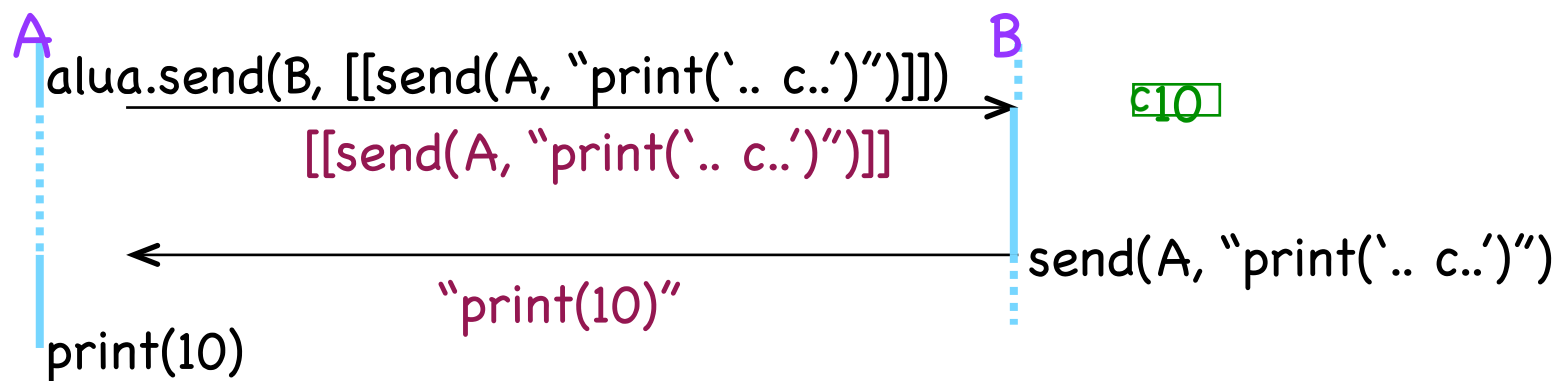
Event-Driven Programming

- **One execution stream: no CPU concurrency.**
- **Register interest in events (callbacks).**
- **Event loop waits for events, invokes handlers.**
- **No preemption of event handlers.**
- **Handlers generally short-lived.**



orientação a eventos: ALua

- assincronismo e distribuição geográfica
- ALua: sistema para programação distribuída
 - baseado em Lua
 - chegada de mensagem é um evento
 - tratamento é execução da mensagem recebida



alua.inf.puc-rio.br



co-rotinas em Lua

```
function boba ()
  for i=1,10 do
    print("co", i)
    coroutine.yield()
  end
end
co = coroutine.create(boba)

coroutine.resume(co) -> co 1
coroutine.resume(co) -> co 2
...
coroutine.resume(co) -> co 10
coroutine.resume(co) -> nada... (acabou)
```

*transferência de controle explícita!
Menos problemas com condições de corrida!*



Referências

- notas de aula Ihor Kuz, Manuel M. T. Chakravarty & Gernot Heiser (intro-notes.pdf, na página do curso*)
- J. Ousterhout. Why threads are a bad idea (for most purposes)
- capítulo de co-rotinas - livro de Lua
 - Roberto Ierusalimschy. Programming in Lua. lua.org, 2006.
 - » disponível na secretaria do DI
 - » 1a edição disponível em www.lua.org/pil/

*inf.puc-rio.br/~noemi/sd-09/

