# 4
# Modelo de Concorrência com Eventos em Lua

O GSD-PUC tem investigado um modelo simples de eventos, usando o ALua, para o desenvolvimento das aplicações distribuídas. Um dos objetivos do estudo é analisar o potencial e as dificuldades desse modelo. Mesmo com sua flexibilidade e a utilização de co-rotinas para auxiliar no tratamento dos eventos, existem atividades em que a disponibilidade de multitarefa poderia contribuir no desenvolvimento da aplicação.

Atividades com utilização freqüente de chamadas bloqueantes ou muito tempo de processamento não se enquadram bem no modelo de eventos, e o mais indicado seria disparar tais atividades em uma linha de execução paralela para manter o fluxo principal do programa livre. Para as aplicações que desejam tirar proveito de vários núcleos de processamento (computadores desse tipo estão cada vez mais difundidos), a programação multitarefa é necessária.

Neste capítulo descrevemos as características de um modelo para disponibilizar multitarefa juntamente com orientação a eventos em Lua. Temos como motivação ter uma forma mais simples de desenvolver aplicações concorrentes do que a programação clássica com lock e acreditamos que os eventos podem contribuir para isso.

Processos e threads com memória compartilhada são duas opções bem conhecidas para programação concorrente. Desenvolvimento com vários processos é conhecido desde a criação dos sistemas operacionais multitarefas, mas programação com threads se tornou mais popular nos últimos anos.

Na próxima seção, discutimos um pouco sobre as características de Lua com relação à concorrência e ambientes multithreading. Também apresentamos algumas bibliotecas para concorrência na linguagem, mostrando suas características e decisões de projeto.

## 4.1
## Concorrência em Lua

Programas desenvolvidos com as linguagens Lua e C em conjunto podem ter Lua desempenhando um papel mais central, com diversas funções providas por C sendo coordenadas por códigos Lua. Mas podemos ter o contrário, onde Lua é

empregada como um extensão de C. Para prover essa interação, a linguagem Lua permite que praticamente todas as suas funcionalidades estejam acessíveis em C.

A distribuição padrão de Lua interpreta a linguagem em uma máquina virtual (VM) e permite que várias VMs sejam instânciadas dentro de um programa. Cada instância da VM é comumente chamada de *estado Lua*. Os estados Lua não compartilham informações, ou seja, cada um possui o seu próprio conjunto de funções, variáveis e estruturas de controle. Para realizar troca de informações entre estados, é preciso utilizar algum mecanismo externo, geralmente provido por módulos adicionais.

O estado Lua foi projetado para dar suporte a um único fluxo de execução, mas a implementação padrão prevê que a linguagem seja usada em ambientes multithreading. Algumas macros, como `lua_lock` e `lua_unlock`, podem ser redefinidas externamente para compilar Lua com proteção para o uso ordenado em ambiente multithreading. Sem essa proteção, se várias threads em programa C acessam simultaneamente o estado Lua, não há garantia da computação correta. Se essas macros não são fornecidas externamente, a implementação padrão de Lua as define como nulas.

Como apresentamos anteriormente, Lua oferece concorrência cooperativa por meio das co-rotinas. Cada co-rotina possui sua própria pilha de execução e compartilha as variáveis globais com as demais co-rotinas, mas a troca de contexto é feita de forma explícita. No capítulo 3, mostramos como a utilização de co-rotinas auxiliou na construção dos mecanismos de RPC e na criação do sincronizador, facilitando a suspensão e retomada da computação. OiL (Maia et al., 2005; Maia et al., 2006) (implementação CORBA), ConcurrentLua (Chatzimparmpas, 2007) (implementa os conceitos da linguagem Erlang (Armstrong et al., 1996) em Lua) e Xavante (Xavante Web Server, 2004) (servidor web) são exemplos de projetos em Lua que exploram fortemente o mecanismo de co-rotinas para prover concorrência.



### 4.1.1 Lua e Programação Multitarefa

Em alguns cenários, adotar processos para multitarefa é mais conveniente. Pelo seu isolamento, processos são usados como mecanismo de proteção para evitar o comprometimento do programa – falha ou segurança. Entretanto, de uma forma geral, os desenvolvedores preferem threads, principalmente pelo compartilhamento de recursos provido pelo modelo.

As aplicações Lua podem usar as funções `os.execute` (equivalente à `system` de C) e `io.popen` para criar novos processos. Essas funções invocam o interpretador de comando do sistema para executar o comando desejado, por isso estes devem ser montados especificamente para o interpretador disponível – o que pode gerar

problema de portabilidade. A criação de processos também pode ser oferecida por bibliotecas adicionais como a lposix (Figueiredo, 2000), que exporta a função POSIX `fork` para Lua. Usando uma das duas formas, podemos disparar, por exemplo, o interpretador Lua e iniciar a execução de um programa Lua em um novo processo.

O projeto ALua foi desenvolvido para dar suporte à programação de aplicações distribuídas orientadas a eventos. Apesar de o ALua ter sido desenvolvido para aplicações de rede, nada impede que ele seja usado para criação de aplicações em apenas uma máquina. A função `alua.spawn` permite disparar novos processos do sistema operacional para executar um trecho de código Lua passado como parâmetro. A implementação dessa função realiza chamadas a função nativas do sistema operacional para a criação dos processos, como `CreateProcess` no Windows e `fork`/`exec` em UNIX.

Cada processo criado possui o seu próprio identificador e a comunicação entre esses processos é feita via canais TCP. Os eventos no ALua são strings contendo código Lua. A função `alua.send` envia os eventos de forma assíncrona, ela recebe como parâmetro o identificador do processo e o evento a ser transmitido. A aplicação é responsável por montar a string, serializando os dados se necessário.

A utilização de Lua em um ambiente com multiplas threads tem alguns detalhes devido às características de implementação da linguagem. LuaThread (Nehab, 2004) é um projeto que define as macros `lua_lock` e `lua_unlock`, dentre outras, para permitir o uso do estado Lua com multithreading. Ele também disponibiliza um conjunto de funções em Lua para a criação de threads, variáveis de condição e locks. As threads criadas executam sobre o mesmo estado Lua, possibilitando o modelo clássico de desenvolvimento com memória compartilhada.

No entanto, a proteção do estado Lua segue uma abordagem conhecida como *big-lock*, apresentando uma alta granularidade na proteção. Há um lock que engloba todo o estado Lua e obriga que apenas uma thread esteja usando o estado por vez. Os programas Lua usando LuaThread tendem a ter uma baixa concorrência.

Helper Threads (Guerra, 2006) segue uma abordagem de manter o programa Lua como um coordenador de atividades e usar threads para realizar ações que podem bloquear ou tomar muito tempo de processamento. Helper Threads adota um modelo de *bag of tasks*: quando uma thread é criada, ela tem associada uma fila de tarefas a serem processadas e uma fila para fornecer os resultados. As tarefas são escritas em C e não podem acessar o estado Lua diretamente, o que garante a integridade. Helper Threads já disponibiliza a criação de algumas tarefas para manipulação de arquivos e conexões de rede, mas novas tarefas pode ser incluídas por outras bibliotecas (por isso é considerado pelo autor um facilitador). A figura 4.1 mostra um exemplo de Helper Threads que cria uma tarefa para realizar a leitura de

um bloco de dados de um arquivo.

O programa coordenador Lua cria uma tarefa chamando uma função específica exportada pela biblioteca, e adiciona essa tarefa na fila de processamento. Dependendo do tipo da tarefa, a execução pode ser ininterrupta ou, em um dado momento, a thread pode necessitar interagir com o programa coordenador Lua. Em ambos os casos, a tarefa é colocada de volta na fila de resultados e o programa coordenador fica encarregado de recuperar o resultado gerado ou realizar as ações necessárias para que a tarefa volte a executar novamente.

```
-- Cria as filas e uma nova thread associada as mesmas.
tasks = helper.newqueue()
resps = helper.newqueue()
helper.newthread(tasks, resps)

-- Cria uma tarefa para ler um bloco de dados de um arquivo.
f = io.open("file.txt", "r")
task = nb_file.read(f, blksize)
tasks:addtask(task)
```

Figura 4.1: Exemplo de criação de uma thread e tarefa com Helper Threads.

Diferentemente das tarefas implementadas em C de Helper Threads, Lanes (Kauppi, 2007) permite a programação na própria linguagem Lua e explora diversos estados Lua para fornecer concorrência. A biblioteca cria um novo estado Lua, carrega o código nesse estado e dispara uma thread para executá-lo. Assim, cada estado é isolado um do outro, e a comunicação entre eles se dá através de um mecanismo baseado em espaço de tuplas, denominado pelo autor como *linda*.

As funções `send` e `receive` são usadas para interagir com o espaço, respectivamente enviando e recebendo dados. Essas funções têm como primeiro parâmetro um valor que será a chave de registro e procura para os demais valores sendo enviados. Por exemplo, a figura 4.2 mostra trechos de código usados para a comunicação entre duas threads. A thread *A* envia as dimensões de uma janela gráfica que deve ser criada pela thread *B*.

```
-- Thread A
spc = lanes.linda() -- pega referência ao espaço
spc:send("window", 800, 600)

-- Thread B
spc = lanes.linda()
height, width = spc:receive("window")
create_window(height, width)
```

Figura 4.2: Exemplo de comunicação entre threads em Lanes.

Lanes utiliza estados Lua adicionais para manter os valores trocados entre threads através do mecanismo *linda*. Na hora do envio, o remetente copia os valores

do seu estado Lua para um dos estados da *linda*. O destinatário, por sua vez, ao chamar a função de recebimento, copia os valores da *linda* para o seu estado. A figura 4.3 ilustra esse procedimento de troca de informações.

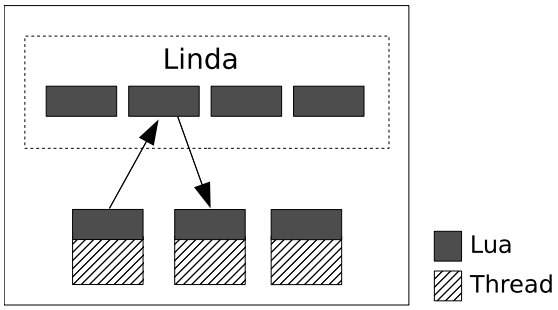


Figura 4.3: Implementação do mecanismo *linda* de Lanes.

A cópia dos valores é feita diretamente entre estados via a API de Lua, evitando a serialização dos valores para um formato intermediário. Lanes usa um conjunto de estados na tentativa de distribuir a carga de acesso e evitar um gargalo na comunicação. As operações de envio e recebimento sobre um mesmo estado Lua dentro da *linda* devem ser protegidas, pois, por padrão, Lua não suporta acesso concorrente. Lanes emprega um lock para cada estado da *linda* para garantir a consistência das operações de leitura ou escrita.

luaproc (Skyrme, 2007) é uma biblioteca que também adota a abordagem de estados Lua isolados, mas flexibiliza a relação entre as threads e os estados. O objetivo é reduzir o número de threads usadas, já que uma thread consome bem mais recursos se comparado com um estado Lua (Skyrme, 2007). Em Lanes, uma thread é associada a um estado Lua e seu tempo de vida depende do código Lua sendo executado. Entretanto, em luaproc, a execução de um estado Lua pode ser suspensa e a thread pode iniciar (ou retomar) a execução de outro estado. A figura 4.4 mostra a relação das threads e estados Lua em Lanes e luaproc.

A biblioteca luaproc possui um conjunto de threads que ficam encarregadas de executar os estados Lua. Podemos criar novos estados Lua para executar novas tarefas sem aumentar o número de threads envolvidas. Mas se necessário, luaproc também permite aumentar ou diminuir a quantidade de threads em tempo de execução.

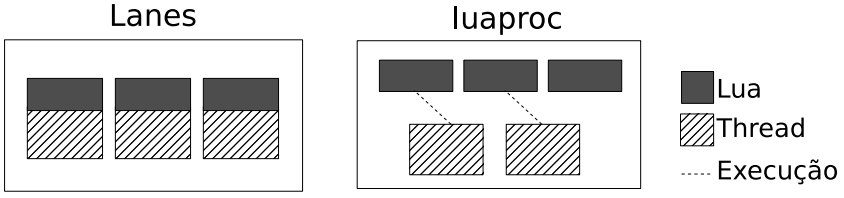


Figura 4.4: Relação entre threads e Lua em Lanes e luaproc.

A comunicação entre os estados é feita por meio de passagem de mensagens em canais síncronos. Isso significa que se o programa em um estado Lua tentar enviar ou receber uma mensagem, ele ficará bloqueado enquanto outro estado não executar a operação complementar. Nesse ponto, em vez de bloquear a thread e esperar um par para que a comunicação se realize, luaproc suspende apenas o processamento do estado e o coloca em uma fila de espera. A thread é liberada e irá processar outro estado Lua disponível.

## 4.2 Eventos Paralelos em Lua

A nossa proposta para a execução paralela de eventos é baseada na utilização de vários estados Lua, onde cada estado terá o seu próprio loop de eventos. Assim como luaproc e Lanes, os estados Lua serão a unidade básica de processamento exportada para o desenvolvedor. De agora em diante utilizaremos o termo *processo Lua* para denominar essa unidade básica, que é composta de um estado Lua executando o seu próprio loop de eventos.

Embora a implementação de Lua tenha proteção para ambientes multithread, ela não é otimizada para uso nesse ambiente. Descartamos então a abordagem usada por LuaThread, pois além de ela obrigar que Lua seja compilada de uma forma diferente da padrão (o que pode causar barreira na adoção), ela tem uma forte tendência de prover baixa concorrência. Iremos acomodar o suporte à multitarefa via biblioteca adicional.

Ao adotarmos a separação dos estados, reduzimos o problema de interferência na computação. Esse isolamento reforça a característica de orientação a eventos de que não há interferência durante o tratamento de um evento. Isso segue a idéia proposta por Ousterhout (Ousterhout, 1996) de tentar isolar o máximo possível o desenvolvedor do uso direto de locks e outros mecanismos de mais baixo nível para lidar com concorrência, ficando a cargo de nossa infra-estrutura lidar com esses mecanismos. A figura 4.5 mostra um esquema do nosso modelo.

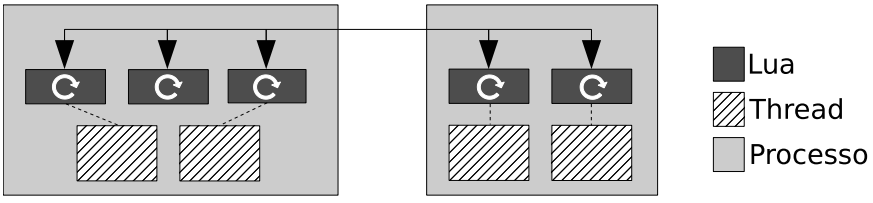


Figura 4.5: Arquitetura proposta para orientação a eventos e concorrência em Lua.

Essa abordagem de isolamento tem semelhanças com Erlang e E (Miller et al., 2005). Ambas linguagens dão uma visão de processos isolados ao programador, permitindo a troca de informações através de troca de mensagens. Evitando estruturas compartilhadas com proteção por locks, essas linguagens evitam vários dos

problemas de concorrência. Além disso, Erlang é bem conhecida pelo seu modelo de processos leves com baixo tempo de criação. Essas características dão aos programas em Erlang uma alta escalabilidade, pois eles conseguem criar um grande número de processos leves que seria difícil de alcançar se fossem usados threads ou processos do sistema operacional. luaproc emprega um conceito parecido, dando visão de que os estados Lua seriam como os processos de Erlang. A implementação de Lua consome mais memória do que os processos de Erlang, mas bem menos que threads (Skyrme, 2007). Assim, achamos que adotar uma relação flexível entre threads e estados Lua em nosso modelo pode representar um melhor aproveitamento de recursos.

Uma primitiva é responsável por disparar novos processos Lua. Ela recebe uma string contendo um trecho de código Lua e, no nível mais baixo, cria algum estado Lua para executá-lo. De fato, não há garantias de que o código será imediatamente executado. Esse processo Lua será colocado na fila de prontos e em algum momento será processado. Algumas bibliotecas básicas podem ser carregadas na inicialização, mas todo o restante é de responsabilidade do trecho de código.

A troca de informações entre os processos Lua será realizada pela emissão de eventos assíncronos. Seguiremos a abordagem do ALua, onde os eventos são strings que serão processados pelo receptor. A aplicação será responsável por construir os seus eventos. Bibliotecas de serialização como Pluto (Sunshine-Hill, 2004) podem ser utilizadas para auxiliar na montagem dos eventos, transformando valores da aplicação em string.

Ao adotarmos estados separados, o mecanismo de comunicação vai cumprir um papel importante. Queremos que o modelo suporte tanto o uso de threads quanto o de processos, mas queremos explorar as características de cada ambiente.

A camada de comunicação ficará responsável por abstrair a localização dos estados, podendo assim adotar formas mais eficientes de entrega das mensagens sem expor isso ao desenvolvedor. Por exemplo, estados dentro de um mesmo processo podem aproveitar o acesso comum à memória e até estruturas de alta concorrência, para a troca das mensagens. De fato, a construção do modelo pode ser dividida em duas partes: concorrência dentro de um processo e interligação dos processos.

Também é importante que o modelo aceite eventos vindos não somente da comunicação dos estados. Módulos adicionais podem gerar eventos que devem ser tratados pela aplicação, como por exemplo temporizadores. Assim como na coordenação, queremos que a interação seja flexível para possibilitar extensões.

No próximo capítulo, apresentaremos a implementação do modelo proposto.