1
Introdução

O uso de *multithreading* tem se popularizado como forma de separar a execução de tarefas concorrentes e de alcançar maior desempenho aproveitando melhor o tempo das CPUs. Multithreading preemptiva com compartilhamento de memória é um modelo muito adotado atualmente para o desenvolvimento. Em aplicações onde há a necessidade de realizar múltiplas tarefas ao mesmo tempo, delegar cada tarefa para uma thread acaba sendo a escolha mais conveniente. O mecanismo de preempção, disponível na maioria dos sistemas operacionais atuais, livra o desenvolvedor da atividade de escalonar manualmente as tarefas, dando uma visão de várias execuções seqüenciais sem interrupção.

No entanto, a programação com threads não é uma tarefa fácil. O uso dos recursos compartilhados deve ser coordenado, pois o acesso concorrente aos mesmos, na maioria dos casos, gera inconsistência na aplicação. *Locks* e semáforos são mecanismos bem conhecidos para promover a coordenação, criando um acesso exclusivo a pontos críticos do sistema (Andrews, 2000; Silberschatz et al., 2008). Uma das maiores dificuldades e fonte de erros em programas multithreading está em empregar incorretamente os mecanismos de coordenação. Condições de corrida são difíceis de se identificar, e as ferramentas de depuração muitas vezes não são úteis, pois elas interferem no escalonamento normal das threads e não conseguem reproduzir o problema. O suporte a threads também não é padronizado, gerando problemas de portabilidade e de comportamento de acordo com a plataforma de execução. Além disso, a própria necessidade de preempção deve ser avaliada, pois a troca de contexto gera custos computacionais (salvar os registradores, invalidação de *cache*) e, dependendo da estrutura da aplicação, essa alternância entre threads pode ser mais custosa que a simples realização das tarefas em seqüência.

Devido a essas complicações, o modelo de desenvolvimento orientado a eventos foi apontado por alguns como uma boa alternativa na criação de aplicações (Ousterhout, 1996; Dabek et al., 2002; Lee, 2006). Nesse modelo, a tarefa é realizada por um ou mais eventos, e um *loop* principal fica responsável por receber e despachar esses eventos. Quando um evento é recebido, o fluxo de execução é desviado para o seu tratamento, fazendo com que o controle só retorne ao loop quando o processamento do evento chegar ao fim. Vários dos problemas associados à concorrência

das threads são evitados ao darmos suporte à execução de um evento por vez.

O desenvolvimento orientado a eventos também apresenta limitações. Como há apenas um loop de eventos, eventos muito longos monopolizam o processamento e não abrem espaço para que novos eventos sejam executados. Dessa forma, a recomendação é que processamentos longos sejam quebrados em eventos menores. Chamadas bloqueantes também devem ser evitadas, dando preferência à forma assíncrona (se disponível) para evitar que o fluxo de execução seja retido por muito tempo. Funções de *callback* podem ser associadas às funções assíncronas, ficando responsáveis por receber o valor de retorno e retomar o processamento.

A criação de aplicações usando eventos é parecida com a programação em máquina de estados. Cada evento processado modifica o estado atual da aplicação, levando a um novo estado global. Essa forma de programação nem sempre é intuitiva e muitos desenvolvedores preferem o modelo com fluxo mais linear da programação com threads, pois à medida que o número de eventos e callbacks aumenta, a legibilidade e a manutenibilidade do programa se tornam mais difíceis (von Behren et al., 2003). Além disso, tarefas compostas por vários eventos sofrem de um processo conhecido como *stack ripping* (Adya et al., 2002), onde os valores locais de cada evento devem ser mantidos em áreas globais para que o estado da computação seja preservado, pois ao término de cada evento a pilha de execução é desfeita e os valores intermediários são perdidos.

Alguns trabalhos vêem investigando o uso de multithreading cooperativa como forma de facilitar a gerência do estado dos eventos, não introduzindo os problemas de concorrências de multithreading preemptiva (Behren et al., 2003; Fischer et al., 2007). Em multithreading cooperativa, apenas uma thread está em execução por vez, e a troca de contexto entre as threads é feita de forma explícita. Uma das vantagens é que o estado de execução é mantido quando a thread é suspensa, facilitando a retomada do processamento. Outra vantagem é a garantia de que, durante o processamento de um evento, não ocorrerá interferência. Isso reduz o problema de concorrência a pontos bem conhecidos, mais precisamente, aos pontos onde a thread explicitamente cede o controle do processador. Mesmo assim, não resolve inteiramente o problema. Ainda temos a necessidade de mecanismos de sincronização e coordenação para garantir a consistência nas aplicações.

Existem algumas considerações que devem ser levadas em conta ao usarmos multithreading cooperativa. Chamadas a funções bloquearão todo o processo, esperando o término da chamada. Dado que a passagem de controle é explícita, outra thread não terá a oportunidade de assumir o processador. Além disso, se um evento realiza computação intensa, não temos pontos oportunistas para a suspensão. Nesse caso, o programador pode efetuar a liberação do processador em pontos arbitrários, mas isso não nos dá a garantia que a troca de contexto terá benefícios, pois

ao retornarmos o fluxo para o loop de eventos, este pode não ter outro evento para tratar, retomando a execução do evento que acabou de ceder a vez. Multithreading cooperativa também não permite que as aplicações tirem proveito de máquinas multiprocessadas, cada vez mais difundidas.

Esses aspectos nos levam de volta a abordagem concorrente na construção de aplicações. Porém, é importante estudar como isso pode ser feito sem cair novamente nos problemas que inicialmente motivaram o uso de orientação a eventos. Como apontado em (Ousterhout, 1996), um caminho para isso é isolar a questão de concorrência o máximo possível do restante da aplicação para que o desenvolvedor não tenha que lidar com ela diretamente.

1.1 Eventos em Lua

O grupo de Sistemas Distribuídos do Departamento de Informática da PUC-Rio (GSD-PUC) tem investigando nos últimos anos o modelo orientado a eventos para o desenvolvimento de aplicações distribuídas em Lua (Leal et al., 2003; Rossetto et al., 2004; Rodriguez e Rossetto, 2008), tendo o ALua (Ururahy e Rodriguez, 1999; ALua: asynchronous distributed programming in Lua, 2004) como infra-estrutura para a troca de eventos. Para introduzir a possibilidade de trabalhar com várias linhas de execução, evitando a complexidade de multithreading preemptiva, parte da investigação explorou multithreading cooperativa oferecida por Lua (Rodriguez e Rossetto, 2008).

Temos então como resultado um ambiente de processos distribuídos com diversas linhas de execução por processo. A coordenação e sincronização dessas várias linhas exigem mecanismos flexíveis que possam se adaptar aos requisitos das aplicações, permitindo que o programador utilize a forma mais adequada para cada situação. Algumas linguagens embutem mecanismos de coordenação, fornecendo modelos consistentes com a linguagem, mas pré-determinados. Outra solução é fornecer os mecanismos através de bibliotecas, que permitem maior flexibilidade na escolha, mas nem sempre são bem integrados à linguagem.

Investigamos, neste trabalho, como características da linguagem podem ser utilizadas para prover flexibilidade na escolha e combinação de mecanismos de coordenação. Essas características podem ajudar, por exemplo, a preencher a lacuna no uso de bibliotecas, incorporando-as uniformemente na linguagem, ou compor novos mecanismos a partir de existentes.

Uma segunda linha de investigação se refere ao modelo de concorrência com orientação a eventos em Lua. Apesar de o objetivo inicial do projeto ALua ser o de fornecer um modelo de programação sem concorrência para aplicações distribuídas, nossa experiência de uso do ALua, aliada à evolução das arquiteturas

de processadores, nos fizeram observar que em determinados casos é desejável ter um suporte à multitarefa. Desejamos combinar o modelo orientado a eventos com concorrência sem trazer de volta os problemas a ela associados.

Existem duas opções bem conhecidas para multitarefa em aplicações: processos com a utilização dos mecanismos de IPC para intercomunicação e o modelo de threads com compartilhamento de memória (como threads POSIX). A maioria dos sistemas operacionais modernos permite a criação de processos independentes e fornece algum mecanismo para que eles se comuniquem. Algumas aplicações podem tirar proveito desse encapsulamento para criar um ambiente de execução confiável, onde algumas de suas partes seriam executadas em processos isolados.

Por estarem dentro do mesmo processo as threads podem tirar proveito de uma troca de informações mais direta. Em vez de realizar a serialização dos dados e enviá-los por canais de comunicação, elas podem se beneficiar de estruturas compartilhadas para o envio de informações a outras threads dentro do mesmo processo. Por outro lado, esse mesmo compartilhamento é o que traz a maior complexidade no desenvolvimento de aplicação com multithreading. Vamos explorar características de Lua para esconder do programador essa complexidade.

1.2 Objetivos

O primeiro objetivo deste trabalho é investigar como oferecer concorrência no modelo de eventos do ALua, para explorar a paralelização de tarefas. Os trabalhos do GSD-PUC com o ALua tinham um foco voltado para a distribuição. Nossa contribuição é para flexibilizar mais a infra-estrutura, disponibilizando, além da criação de novos processos, multithreading. Com isso, queremos investigar melhor a programação multitarefa com os eventos em Lua. Mas por outro lado, temos o desafio de aproveitar as vantagens de threads (como o compartilhamento de recursos), tentando evitar ao máximo os problemas de concorrência citados anteriormente.

Com o modelo de eventos com concorrência, criamos então a possibilidade de várias linhas de execução com eventos, e se considerarmos ainda a utilização de multithreading cooperativa e processos remotos, teremos diversos fluxos diferentes no sistema. A interação dessas partes pode se dar de diversas maneiras, e elas devem ser coordenadas para a realização das atividades propostas pelo sistema. Surge uma demanda por mecanismos de coordenação flexíveis, que permitam ao programador utilizar a forma mais adequada para o desenvolvimento de sua aplicação. Assim, o segundo objetivo deste trabalho é investigar como as características da linguagem de programação podem contribuir para que mecanismos de coordenação e sincronização possam ser criados, combinados e integrados uniformemente na linguagem.

O capítulo 2 apresenta uma discussão maior sobre orientação a eventos e concorrência. No capítulo 3, investigamos como as características de linguagens dinâmicas podem ajudar na criação de mecanismos de coordenação adaptáveis, os quais podem ser combinados e estendidos. O capítulo 4 apresenta um modelo de eventos com concorrência para Lua e sua implementação é descrita no capítulo 5. No capítulo 6, apresentamos dois casos de uso do modelo de eventos proposto e, no capítulo 7, apresentamos as conclusões deste trabalho.