6
Co-rotinas completas e programação concorrente

Originado no contexto de sistemas operacionais, há mais de três décadas [20, 38, 39], o uso de concorrência é cada vez mais presente no desenvolvimento de sistemas e aplicações em diferentes domínios. Exemplos de cenários onde a programação concorrente é observada incluem serviços WEB e aplicações distribuídas em geral, sistemas de interface gráfica, aplicações paralelas, jogos, simulação de eventos discretos, e outros tipos de aplicações onde a decomposição em tarefas paralelas — isto é, executadas em concorrência real ou simulada — visa atender uma combinação de requisitos como desempenho, responsividade e simplicidade de projeto.

Atualmente, o modelo conhecido como *multithreading* é praticamente um padrão para o desenvolvimento de aplicações concorrentes. A maioria das linguagens *mainstream* modernas, como Java [56], C# [3], Perl [87] e Python [63], e bibliotecas largamente utilizadas como pThreads [67], provêem *threads* como construção básica de concorrência. A noção de que *threads* representam uma ferramenta mais simples e eficiente para o suporte à implementação de programação concorrente contribuiu de forma decisiva para virtual abandono do interesse em co-rotinas como construção de concorrência.

Multithreading envolve tipicamente a execução de tarefas concorrentes que compartilham memória e são sujeitas a preempção — isto é, à perda involuntária do controle do processador. Essas características são essenciais no contexto que originou a programação concorrente — o desenvolvimento de sistemas operacionais — onde requisitos de responsividade e desempenho são extremamente rigorosos. Entretanto, os requisitos de aplicações concorrentes são, em geral, bastante diferentes dos requisitos de um sistema operacional. Além disso, programadores de aplicações, diferentemente de desenvolvedores de sistemas operacionais, são muitas vezes relativamente inexperientes, ou pouco expostos aos problemas e soluções relacionados ao uso de concorrência. Nesse cenário, o uso do modelo de *multithreading* é claramente inadequado, e pode neutralizar os benefícios potenciais da pro-

gramação concorrente, produzindo aplicações desnecessariamente complexas, incorretas e de baixo desempenho.

Nos últimos anos, os diversos problemas relacionados ao uso de *multithreading* têm motivado a investigação e proposta de ambientes baseados em modelos de concorrência alternativos, destacando-se a programação orientada a eventos [47, 89, 7, 16, 91] e, em menor escala, a gerência cooperativa de tarefas [1, 5]. Ao evitar a união de mecanismos de preempção e memória compartilhada, esses modelos permitem o desenvolvimento de aplicações menos complexas, de melhor desempenho, e com maiores garantias de correção.

Um outro modelo alternativo, usualmente desconsiderado, é o modelo de *processos* — um modelo de concorrrência baseado na interação de tarefas através de troca de mensagens. O pouco investimento em implementações eficientes desse tipo de modelo, especialmente em mecanismos para a interação entre processos em uma mesma máquina, é em parte responsável pela rejeição ao uso de processos para a implementação de aplicações concorrentes, privilegiando a adoção de modelos baseados em memória compartilhada, particularmente *multithreading*. Entretanto, o modelo de processos é bastante adequado para o desenvolvimento de aplicações naturalmente decompostas em módulos fracamente acoplados. Além de favorecer uma melhor estruturação desse tipo de aplicações, o uso de processos facilita a transposição dessas aplicações para ambientes distribuídos. O modelo de processos constitui também uma opção conveniente para ambientes multiprocessadores, especialmente quando combinado a um dos outros modelos alternativos.

Neste capítulo, nós analisamos benefícios e desvantagens associados a cada tipo de modelo de concorrência, justificando a adoção de modelos alternativos a *multithreading*. Mostramos também que co-rotinas completas assimétricas constituem uma construção bastante apropriada como suporte a modelos baseados em gerência cooperativa de tarefas e programação orientada a eventos e, portanto, mais adequadas que *threads* como uma construção básica de concorrência para ambientes de memória compartilhada.

6.1 Análise de modelos de concorrência

Apesar de introduzir novas oportunidades, o uso de concorrência introduz também problemas não encontrados no projeto, desenvolvimento e manutenção de aplicações seqüenciais. A escolha de um determinado modelo de

concorrência influi consideravelmente na minimização, ou potencialização, desses problemas.

6.1.1 Caracterização de modelos de concorrência

Para efeito da análise que vamos apresentar, utilizamos uma caracterização de modelos de concorrência baseada na combinação de três mecanismos básicos:

Mono/multi-tarefa: O uso de concorrência envolve a decomposição de uma aplicação em um conjunto de tarefas, ou fluxos de controle, independentes. Quando todas as tarefas executam sem transferência de controle até o seu final, em qualquer instante uma única linha de execução é definida. Denominamos este tipo de mecanismo *mono-tarefa*. Por outro lado, quando diversas tarefas executam simultaneamente, em um determinado instante múltiplas linhas de execução poderão estar definidas. Denominamos esse mecanismo *multi-tarefa*.

Escalonamento: Um modelo multi-tarefa implica em um entrelaçamento de diferentes linhas de execução; a política de escalonamento utilizada determina de que formas esse entrelaçamento poderá ocorrer. Quando essa política é *preemptiva* — seja pela adoção de mecanismos de prioridade, de *time slicing*, ou ambos — uma alternância entre tarefas pode ocorrer a qualquer momento. Nesse caso, o entrelaçamento de linhas de execução é arbitrário, ou não-determinístico. Quando o escalonamento é *não preemptivo*, uma transferência de controle ocorre somente em pontos bem definidos da execução de uma tarefa. Essa transferência de controle pode ser implícita ou explícita. Uma transferência implícita ocorre, por exemplo, quando uma tarefa inicia uma operação de entrada e saída, ou quando se coloca à espera de uma condição. Uma transferência explícita ocorre quando uma tarefa cede voluntariamente o controle do processador.

Interação entre tarefas: A decomposição de uma aplicação em um conjunto de tarefas envolve a necessidade de cooperação entre elas. Essa cooperação pode ser obtida através de dois mecanismos básicos: *memória compartilhada* ou *troca de mensagens* [2]. Um mecanismo de memória compartilhada permite que tarefas independentes troquem informações através do uso de estruturas de dados armazenadas em

um espaço de endereçamento comum. Um mecanismo de troca de mensagens permite a transferência de informações através da provisão de *canais de comunicação* e de operações básicas para envio e recepção de mensagens através desses canais.

Principais modelos de concorrência

Em nossa abordagem, um modelo de concorrência é o resultado da combinação dos mecanismos descritos. Entretanto, somente algumas dessas combinações resultam em modelos relevantes. Esses modelos, e exemplos típicos de suas implementações, são identificados a seguir:

- mono-tarefa, com memória compartilhada: *orientação a eventos*
- multi-tarefa, com preempção e memória compartilhada: *multithreading*
- multi-tarefa, sem preempção e com memória compartilhada: *gerência cooperativa de tarefas*
- multi-tarefa, com/sem preempção e com troca de mensagens: *processos*

Uma consideração importante diz respeito à categorização do modelo conhecido como *multithreading*. As implementações mais conhecidas e utilizadas desse modelo — como a de Java e pThreads — não oferecem qualquer garantia quanto à política de escalonamento adotada. Para garantir a portabilidade de uma aplicação, seu desenvolvedor deve então a assumir o comportamento de um modelo preemptivo, e implementar soluções para os problemas associados a esse comportamento, mesmo que em algumas plataformas a política de escalonamento não seja preemptiva.

6.1.2 Complexidade de Programação

Uma noção bastante comum é a de que quanto mais distante do modelo seqüencial — mais familiar, ou intuitivo, à maioria dos desenvolvedores — mais difícil é o projeto e programação de uma aplicação concorrente.

Na programação orientada a eventos, uma aplicação é tipicamente estruturada como um conjunto de tarefas tratadoras de eventos (*event handlers*), ativadas por um *loop* à medida que os eventos correspondentes são observados. Um dos maiores obstáculos ao uso desse modelo é o estilo

de execução tipicamente "reativo" das tarefas, e a conseqüente inversão no fluxo de controle da aplicação. Esse estilo de programação simplifica, contudo, o projeto e programação de aplicações e sistemas inerentemente assíncronos, como, por exemplo, sistemas de interface gráfica e aplicações distribuídas [86, 58, 7, 76].

Para garantir um nível adequado de concorrência à aplicação, modelos mono-tarefa, como o de orientação a eventos, requerem tarefas compostas por pequenos trechos de código, necessariamente não bloqueantes. Em algumas situações, essa exigência pode implicar na quebra de uma tarefa em duas ou mais partes, e, consequentemente, na necessidade de preservação e recuperação de contextos de execução — isto é, na gerência "manual" da pilha de execução [1]. Mecanismos de continuações de primeira classe e co-rotinas (como veremos na Seção 6.3) provêem suporte para a manutenção do estado de tratadores de eventos, simplificando a implementação de aplicações que adotam modelos mono-tarefa.

A aparência de programação sequencial é mais facilmente obtida em modelos multi-tarefa. Nesses modelos, a granularidade das tarefas é significativamente maior, pois é a alternância entre elas que determina o nível de concorrência da aplicação. Sob esse aspecto, modelos *preemptivos* são usualmente considerados mais simples, pois essa alternância é definida pelo escalonador. Por outro lado, como veremos a seguir, a união de mecanismos de preempção e de memória compartilhada potencializa os demais problemas enfrentados por desenvolvedores de aplicações concorrentes. Dessa forma, a aparente maior simplicidade de modelos que apresentam essa união pode, na verdade, envolver um nível de complexidade de programação considerável.

O uso de modelos de concorrência onde a interação entre tarefas é baseada em canais de comunicação simplifica o projeto e desenvolvimento de aplicações onde as tarefas são fracamente acopladas, favorecendo uma modularização adequada dessas aplicações. Essa característica pode ser observada, por exemplo, em aplicações paralelas, um cenário onde espaços de tuplas [32] são bastante utilizados. Apesar de esse mecanismo ser geralmente descrito como uma implementação de memória compartilhada, a leitura e escrita de tuplas é realizada somente através de operações explícitas para o envio e recepção de informações (`out`, `in`), que implementam um acesso serializado — ou seja, não simultâneo — às informações compartilhadas. Dessa forma, podemos categorizar espaços de tuplas como um mecanismo baseado em troca de mensagens. Veremos mais tarde que modelos baseados em troca de mensagens são também apropriados para a implementação de arquiteturas baseadas em computação por estágios, uma solução conveni-

ente para ambientes multi-processadores.

Interações entre tarefas fortemente acopladas, que envolvem o compartilhamento de estruturas de dados, são em geral mais simples e mais eficientes com o uso de mecanismos de memória compartilhada. O uso de memória compartilhada favorece também a preservação da aparência de uma programação seqüencial convencional. Além disso, soluções que envolvem a transferência de estruturas de dados compartilhadas em menssagens podem gerar problemas de inconsistência e serialização.

6.1.3 Garantia de correção

A maior dificuldade introduzida pelo uso de concorrência é garantir a *correção* da aplicação. Essa correção está associada ao atendimento de dois tipos de propriedades: *safety* e *liveness* [2].

Propriedades de *safety* garantem que uma aplicação jamais alcance um estado irremediavelmente inconsistente ou incorreto. Essa garantia exige, em primeiro lugar, a coordenação do acesso aos recursos compartilhados pelas diversas tarefas, evitando que uma interferência entre elas comprometa a consistência do estado da aplicação. Essa coordenação envolve, tipicamente, o uso de mecanismos de sincronização por *exclusão mútua* e por *condições* como semáforos, *locks*, monitores e guardas [9, 6, 2, 56].

Uma outra exigência relacionada à correção da aplicação é a ausência de *deadlocks*, situações onde um conjunto de tarefas, ciclicamente dependentes, é bloqueado à espera de condições que jamais ocorrerão. A ocorrência de um *deadlock* pode ser definida como um estado inconsistente da aplicação e, portanto, como uma falha no atendimento a propriedades de *safety*. Alguns autores porém, como Lea [56], consideram a ocorrência de *deadlocks* como falhas no atendimento a propriedades de *liveness*.

Excluindo-se as situações de *deadlock*, o conceito de *liveness* diz respeito à justiça (*fairness*) na alocação do processador entre as tarefas. Uma alocação injusta pode levar a uma situação onde uma ou mais tarefas jamais conseguem o controle do processador (*starvation*). Contudo, qualquer situação que impeça uma ou mais tarefas de progredir adequadamente pode ser considerada uma falha de justiça.

Em modelos mono-tarefa, a coordenação do acesso a recursos compartilhados é trivial, pois cada tarefa executa sem cessão de controle até o seu final. Além disso, como não há bloqueio de tarefas, situações de *deadlock* jamais ocorrerão. Contudo, se tratadores de eventos são decompostos em di-

versas partes, mecanismos de sincronização podem ser necessários quando tratadores diferentes compartilham recursos. Nesse caso, o comportamento é semelhante ao de um modelo multi-tarefa não preemptivo, discutido mais adiante.

A justiça na alocação do processador em modelos mono-tarefa depende, basicamente, de dois fatores. O primeiro está relacionado à granularidade das tarefas, que deve estar de acordo com os requisitos específicos da aplicação. O segundo fator diz respeito ao escalonamento das tarefas — isto é, a determinação do próximo evento a ser tratado. Neste caso, se algum mecanismo de prioridades for adotado, cuidados especiais devem ser tomados para evitar situações de *starvation*.

A ausência de problemas de consistência e de *deadlocks* simplifica muito a depuração de aplicações concorrentes que adotam um modelo mono-tarefa. Comportamentos inadequados dizem respeito, apenas, a problemas de responsividade, cuja detecção é elementar. Cenários de execução são facilmente reproduzidos, pois dependem exclusivamente da ordenação de eventos. Dessa forma, a depuração desse tipo de aplicações não oferece grandes dificuldades.

Em modelos multi-tarefa não preemptivos, como os momentos onde uma tarefa cede o controle do processador são bem definidos, a manutenção da consistência da aplicação é bastante facilitada. Muitas vezes, mecanismos de sincronização podem ser dispensados ou simplificados, o que diminui a possibilidade de ocorrência de *deadlocks*. Entretanto, para que essa estratégia não comprometa a consistência da aplicação, é indispensável que a semântica das operações que podem acarretar uma cessão implícita de controle — por exemplo, operações de entrada e saída — seja bem definida.

Na ausência de preempção, o sequenciamento de tarefas é determinístico. Dessa forma, cenários de execução são mais facilmente reproduzidos, o que simplifica a deteção e correção de problemas.

O uso de uma política de escalonamento não preemptiva pode, contudo, reduzir o nível de concorrência, ou justiça, da aplicação. Uma tarefa que executa durante muito tempo até ceder o controle, implícita ou explicitamente, impede o progresso das demais tarefas. Neste caso, o nível de justiça adequado pode ser obtido através da redução da granularidade "interna" da tarefa, com a inserção de operações de cessão voluntária de controle. Essa estratégia pode introduzir alguma complexidade na aplicação. Por outro lado, problemas de justiça não são difíceis de identificar, e, como vimos, tem solução trivial.

Modelos baseados em troca de mensagens em geral não oferecem

grande dificuldade de coordenação. Em primeiro lugar, o próprio mecanismo de troca de mensagens oferece facilidades de sincronização (Lauer e Needham [55], por exemplo, demonstram a dualidade desse mecanismo e mecanismos de sincronização como monitores). Além disso, esse modelo favorece a decomposição de uma aplicação em tarefas fracamente acopladas, o que reduz o uso de recursos compartilhados. Essa característica simplifica a sincronização de tarefas, reduzindo a probabilidade de erros de inconsistência e de ocorrência de *deadlocks*.

Com respeito à justiça na alocação do processador, o uso de modelos preemptivos oferece alguma facilidade, pois a responsabilidade pela distribuição de recursos é delegada ao mecanismo de escalonamento. Contudo, situações de *starvation* podem ocorrer, especialmente quando mecanismos de prioridade são adotados.

Em modelos preemptivos baseados em memória compartilhada, a coordenação de tarefas é muito mais complexa [9]. Mesmo a semântica de construções simples da linguagem, como o incremento de uma variável compartilhada, pode ser afetada [56]. Erros sutis resultantes da dificuldade em identificar as regiões críticas de uma tarefa são bastante comuns, e, na maioria das vezes, muito difíceis de detetar.

A maior preocupação com a consistência do estado da aplicação, e a dificuldade em delimitar adequadamente suas regiões críticas, provoca um uso intenso — e muitas vezes indiscriminado — de mecanismos de sincronização. Dessa forma, o número de *deadlocks* potenciais aumenta significativamente. Apesar de exaustivamente exploradas na literatura, soluções que visam garantir a consistência do estado da aplicação evitando, ao mesmo tempo, a ocorrência de *deadlocks* são de difícil compreensão e implementação, mesmo para desenvolvedores com grande experiência.

Finalmente, o não-determinismo no sequenciamento de tarefas em modelos preemptivos dificulta, ou mesmo impede, a reprodução de cenários de execução. Como aplicações baseadas em modelos que combinam preempção com o uso de memória compartilhada são bem mais suscetíveis a problemas de correção, um esforço consideravelmente maior é dispendido na depuração dessas aplicações.

6.1.4 Desempenho

Além de influenciar a complexidade de projeto e a dificuldade em garantir a correção de uma aplicação, as características de um modelo

de concorrência interferem diretamente em seu desempenho. O uso de mecanismos de sincronização, a gerência de múltiplos contextos de execução e a política de escalonamento adotada introduzem *overheads* que podem prejudicar o atendimento a requisitos como disponibilidade, tempo de resposta e escalabilidade.

De forma geral, modelos mono-tarefa oferecem um bom desempenho e escalabilidade [47, 16]. Essa característica deve-se à inexistência de *overheads* decorrentes de trocas de contexto, ao uso de memória compartilhada e à ausência de mecanismos de sincronização.

O desempenho de uma aplicação concorrente é significativamente afetado por mecanismos de sincronização. O uso intenso desses mecanismos, além de envolver um custo de processamento, aumenta a probabilidade de ocorrência de situações de contenção, reduzindo o nível de concorrência da aplicação. Dessa forma, problemas de desempenho são muito mais facilmente observados em modelos multi-tarefa preemptivos que utilizam memória compartilhada, como *multithreading* [77]. Modelos não preemptivos ou baseados em mecanismos de troca de mensagens podem oferecer melhores resultados [1, 89, 54].

Em modelos multi-tarefa, trocas de contexto são inevitáveis. Entretanto, o custo associado é maior quando uma política de escalonamento preemptiva é adotada, pois a alternância de tarefas é em geral muito mais frequente. Além disso, a própria implementação de mecanismos de *time slicing* ou prioridades representa um adicional de processamento. Para garantir um nível de desempenho adequado, muitas vezes é necessário limitar o número de tarefas simultâneas [77, 47]. Essa necessidade é maior quando a implementação de um modelo de concorrência é baseada em mecanismos oferecidos pelo sistema operacional, pois o custo de criação de novas tarefas pode ser significativo.

6.2 Gerência cooperativa de tarefas

Gerência cooperativa de tarefas (*cooperative task management*) é o melhor exemplo de um modelo de concorrência multi-tarefa não preemptivo baseado em memória compartilhada. Na maioria das situações, a gerência cooperativa pode substituir com vantagens ambientes de *multithreading*, pois além de preservar a aparência de programação sequencial, esse tipo de modelo minimiza a necessidade de sincronização, favorecendo o desenvolvimento de aplicações mais simples e menos suscetíveis a incorreções.

```
tasks = {}  -- lista de tarefas vivas

-- cria uma tarefa
function create_task(f)
  local co = coroutine.wrap(function()
               f()
               return "task ended" -- sinaliza fim da tarefa
             end)

  -- insere a nova tarefa na lista
  table.insert(tasks, co)
end

-- escalonador
function dispatcher()
  while true do
    local n = table.getn(tasks)  -- número de tarefas vivas
    if n == 0 then break end
    for i = 1, n do
      local res = tasks[i]()     -- reativa tarefa
      if res == "task ended" then
        table.remove(tasks, i)   -- tarefa terminou
        break
      end
    end
  end
end
```

Figura 6.1: Gerência cooperativa de tarefas com co-rotinas assimétricas

Ambientes de gerência cooperativa de tarefas podem ser implementados com facilidade a partir de mecanismos de co-rotinas completas assimétricas. Assim como uma *thread*, uma co-rotina representa uma unidade de execução independente, associada a um estado local privado, compartilhando dados e outros recursos globais com outras co-rotinas. Porém, enquanto o conceito de uma *thread* é tipicamente associado a um escalonamento preemptivo — ou seja, a cessões involuntárias de controle — o escalonamento de co-rotinas é essencialmente cooperativo, pois uma co-rotina deve explicitamente suspender sua execução para permitir que outra co-rotina possa executar.

A Figura 6.1 apresenta uma implementação de um ambiente de gerência cooperativa de tarefas baseado no mecanismo de co-rotinas completas assimétricas provido pela linguagem Lua (descrito na Seção 3.3). Nessa implementação, uma tabela (`tasks`) representa a lista de tarefas vivas, armazenando as referências para as co-rotinas correspondentes. A

função `create_task` é responsável pela criação de novas tarefas. Ela recebe como parâmetro uma função que implementa a tarefa, cria uma co-rotina cujo corpo invoca essa função e insere a referência para essa co-rotina na lista de tarefas vivas.

A função `dispatcher` implementa o escalonador de tarefas. Esse escalonador é simplesmente um *loop* que itera sobre a lista de tarefas, reativando as tarefas vivas (que executam até terminar ou solicitar sua suspensão) e removendo da lista as tarefas que terminam. O término de uma tarefa (ou seja, o término da sua função principal) é sinalizado pelo corpo da co-rotina através do retorno de um valor pré-estabelecido (a *string* `"task ended"`) ao escalonador. A suspensão de uma tarefa é obtida pela invocação da função `coroutine.yield`. Na ausência de um parâmetro, essa função retornará o valor `nil` ao escalonador, que assim saberá que a tarefa correspondente ainda está viva.

Em aplicações que associam uma nova tarefa a cada requisição de serviço, o custo de criação de tarefas deve ser minimizado. Nesse caso, a utilização de um *pool* de co-rotinas reutilizáveis pode ser conveniente. O Apêndice A mostra uma implementação dessa facilidade.

Uma implementação trivial de um ambiente de gerência cooperativo, como a que acabamos de apresentar, pode implicar em um comportamento inaceitável para algumas aplicações. Se, por exemplo, uma co-rotina é bloqueada ao invocar uma operação de entrada ou saída, nenhuma outra co-rotina receberá o controle e, portanto, toda a aplicação será bloqueada até o término dessa operação. Dessa forma, o nível de concorrência da aplicação é afetado, o que pode comprometer consideravelmente o seu desempenho.

Contudo, uma solução bastante simples para esse problema pode ser implementada. As bibliotecas de entrada e saída providas pela maioria das plataformas oferecem usualmente facilidades que permitem associar um tempo máximo de espera (um *timeout*) para o término de uma operação, e aguardar mudanças de estado em um conjunto de recursos, que podem representar arquivos ou canais de comunicação como *sockets*. Exemplos dessas facilidades são as funções `poll` e `select`, providas em plataformas UNIX. Operações de entrada e saída assíncronas oferecidas em plataformas Windows podem também ser utilizadas para implementar essas facilidades.

Com essas facilidades, podemos oferecer uma biblioteca auxiliar de funções de entrada e saída que suspendem a co-rotina em execução se a operação desejada não pode ser imediatamente satisfeita. Nesse caso, um valor que representa o recurso associado à operação invocada é retornado ao chamador da co-rotina. Quando a co-rotina é reativada, a função de entrada

```
function io(resource, <parâmetros adicionais>)
  while true do
    status = basic_io(resource, timeout, <parâmetros>)
    if status == "timeout" then
      coroutine.yield(resource)
    else
      return status
    end
end
```

Figura 6.2: Operação de entrada ou saída não bloqueante

ou saída é retomada, finalizando a operação ou novamente suspendendo a co-rotina se essa operação não se completa no tempo determinado. A Figura 6.2 mostra uma possível estrutura para esse tipo de função.

Podemos então modificar a implementação do escalonador para que quando nenhuma tarefa possa ser ativada — ou seja, quando todas as tarefas vivas estão à espera de operações de entrada ou saída — o escalonador se bloqueie à espera de uma mudança de estado em algum dos recursos envolvidos. No Apêndice A apresentamos essa nova implementação do escalonador. Ierusalimschy [49] apresenta também a implementação de uma aplicação concorrente que utiliza esse tipo de solução para a transferência simultânea de arquivos através do protocolo HTTP.

Mecanismos básicos de sincronização para ambientes de gerência cooperativa baseados em co-rotinas completas assimétricas podem ser oferecidos com bastante facilidade. Um mecanismo de exclusão mútua pode ser obtido através de uma implementação trivial de semáforos binários [19, 6]. Mecanismos de sincronização por condições também têm implementação bastante simples. O Apêndice A mostra também uma possível implementação desses mecanismos.

Comentamos anteriormente que um interesse em ambientes de concorrência baseados em gerência cooperativa de tarefas como uma alternativa a *multithreading* começa a ser observado. Adya et al [1] descrevem implementações desse tipo de ambiente para o desenvolvimento de dois tipos de aplicações concorrentes. Na primeira implementação, um sistema de arquivos distribuído, o ambiente de gerência cooperativa é baseado no mecanismo de *fibers* do Windows [75] (que constitui, na verdade, uma implementação de co-rotinas simétricas). Na segunda implementação, uma aplicação para comunicação *wireless* em dispositivos do tipo PDA, o ambiente de gerência cooperativa é simulado a partir de um mecanismo de *threads* e da associação de uma variável de condição a cada uma das tarefas, e ao escalonador. Para

(re)ativar uma tarefa, o escalonador sinaliza a condição associada à tarefa, e se bloqueia em sua própria condição. Para suspender sua execução, uma tarefa sinaliza a condição do escalonador, e se bloqueia à espera da sinalização de sua condição. É claro perceber que esses procedimentos correspondem também a uma implementação de co-rotinas simétricas. Contudo, o trabalho de Adya et al sequer menciona o termo "co-rotina", e denomina sua proposta de "gerência automática da pilha de execução" (*automatic stack management*).

Behren et al [5] propõem um modelo multi-tarefa não preemptivo — essencialmente, um ambiente de gerência cooperativa de tarefas — como uma alternativa para a implementação de servidores com um alto nível de concorrência. Segundo os autores, esse tipo de modelo é tão eficiente quanto a programação orientada a eventos, porém favorece um estilo de programação mais natural. Para validar sua proposta, implementam um ambiente cooperativo baseado em uma biblioteca de co-rotinas desenvolvida para a linguagem C [85]. Interessantemente, esse trabalho não considera que co-rotinas são a construção básica do ambiente de concorrência, que é caracterizado como uma implementação de *multithreading*.

Ganz, Friedman e Wand [31] descrevem um estilo de programação — denominado *trampolined* — onde um programa é organizado como um *loop* que escalona diferentes computações, cuja execução progride em passos discretos. A aplicação desse estilo de programação para a implementação de *multitasking* corresponde, basicamente, à implementação de gerência cooperativa de tarefas baseada em corotinas completas assimétricas, que apresentamos nesta Seção. De fato, esse trabalho menciona a semelhança desse estilo com o uso de continuações parciais, cuja similaridade com co-rotinas completas assimétricas foi discutida no Capítulo 4. A semelhança com co-rotinas é também mencionada, porém os autores associam o conceito de co-rotinas apenas a co-rotinas simétricas, e, assim, argumentam que a construção de concorrência por eles introduzida (denominada *threads*) não corresponde a co-rotinas.

Modelos de concorrência baseados em gerência cooperativa de tarefas não são, em geral, adequados a ambientes multi-processadores. Nesse tipo de ambiente, modelos que permitem a execução simultânea de tarefas alocadas a diferentes processadores — baseados em *threads* ou em processos — favorecem um maior nível de concorrência para as aplicaçôes, permitindo um melhor aproveitamento dos recursos disponíveis. Conforme discutimos na seção anterior, o modelo de *multithreading*, apesar de ser usualmente considerado mais simples e mais eficiente, envolve uma maior necessidade de me-

canismos de sincronização, o que pode comprometer tanto o desempenho de uma aplicação concorrente como também sua correção. O modelo de processos constitui uma alternativa natural para ambientes multi-processadores, podendo oferecer melhores resultados, especialmente para aplicações que podem ser decompostas em módulos fracamente acoplados.

A combinação de gerência cooperativa de tarefas com modelos baseados em troca de mensagens, como o de processos, representa também uma opção bastante interessante para ambientes multi-processadores. Welsh, Culler e Brewer [89], por exemplo, propõem uma arquitetura para a implementação de servidores WEB onde o atendimento a um serviço é decomposto em uma sequência de estágios. Cada estágio é implementado em um processo separado, e a comunicação entre estágios adjacentes é realizada através de filas de mensagens. Dentro de cada estágio, o atendimento aos "sub-serviços" a ele correspondentes é implementado em um ambiente *multithreaded*. Comparações de desempenho apresentadas pelos autores mostram que em diversos cenários essa arquitetura, denominada SEDA, permite obter resultados significativamente melhores do que arquiteturas baseadas unicamente no modelo de *multithreading*. A natureza assíncrona da comunicação entre estágios é responsável pela caracterização da arquitetura SEDA como um ambiente de orientação a eventos. Contudo, essa arquitetura é, na verdade, um modelo híbrido, baseado na combinação de processos, orientação a eventos e *multithreading*. A implementação de estágios através de gerência cooperativa de tarefas, ao invés de *multithreading*, pode apresentar um desempenho ainda melhor, por evitar, ou reduzir, a necessidade de sincronização intra-estágios. Além disso, como veremos na próxima seção, co-rotinas oferecem um suporte bastante conveniente para a programação orientada a eventos.

Outras formas de decomposição que permitem explorar a combinação de processos e gerência cooperativa de tarefas para um melhor aproveitamento de recursos em ambientes multi-processadores podem ser desenvolvidas. Um exemplo é o atendimento simultâneo a tipos de serviço distintos em processos diferentes; dentro de cada processo, a execução concorrente de tarefas similares pode ser implementada através de um ambiente de gerência cooperativa.

6.3 Programação orientada a eventos

A programação orientada a eventos é um exemplo paradigmático de um modelo de concorrência mono-tarefa. Como observamos anteriormente, esse modelo é bastante adequado a aplicações inerentemente assíncronas, e vem sendo utilizado há bastante tempo para a construção de sistemas de interface gráfica e, mais recentemente, para a implementação de servidores WEB de alto desempenho [47, 89, 16] e de aplicações distribuídas [86, 58, 11, 76].

Uma das maiores dificuldades na implementação desse tipo de modelo é decompor tratadores de eventos em uma ou mais partes, para impedir que tratamentos excessivamente longos comprometam a responsividade da aplicação. Essa decomposição pode ser necessária, por exemplo, para evitar o bloqueio de um tratador de eventos à espera de uma operação de entrada ou saída. Uma situação semelhante ocorre em aplicações distribuídas, quando o tratamento de um evento envolve uma chamada remota síncrona [58, 76]. Nos dois casos, é necessário suspender o tratamento do evento, retomando-o quando a resposta à operação solicitada (um novo evento) for recebida. Para que o tratamento de um evento possa ser suspenso e posteriormente retomado, é necessário algum mecanismo que permita preservar, e restaurar, o contexto de execução desse tratamento.

Lima [58] e Fuchs [30] mostram que continuações de primeira classe provêem suporte a esse tipo de mecanismo. O uso de continuações de primeira classe permite que o contexto de execução de um tratador de eventos — isto é, a *continuação* desse tratador — seja salvo em alguma variável, ou estrutura de dados, para que possa ser recuperado e restaurado quando o resultado que permite o prosseguimento do tratador é obtido.

Rossetto [76] mostra que co-rotinas completas assimétricas também provêem essa facilidade. O trabalho de Rossetto descreve a implementação de um ambiente de desenvolvimento para aplicações distribuídas com mobilidade. Esse ambiente utiliza para os componentes de uma aplicação distribuída um modelo de concorrência baseado em orientação a eventos; a interação entre esses componentes é realizada através de um espaço de tuplas [57]. Operações de escrita de tuplas representam tanto solicitações de serviço como respostas a essas solicitações; essas operações produzem eventos que são enviados aos componentes que registraram seu interesse nas tuplas correspondentes através de operações de leitura. Dessa forma, a inserção de uma tupla associada à solicitação de um determinado serviço,

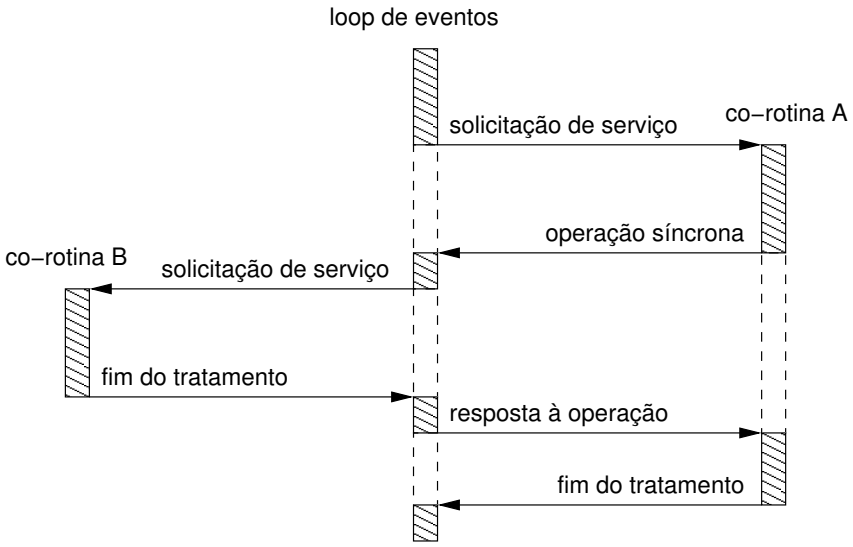

Figura 6.3: Orientação a eventos com suporte de co-rotinas

seguida pela leitura da tupla que contém o resultado desse serviço representa uma operação remota síncrona, e não deve bloquear um componente da aplicação.

Para permitir que tratadores de eventos sejam suspensos enquanto executam uma operação remota síncrona, a execução desses tratadores é realizada por co-rotinas completas assimétricas, como mostra a Figura 6.3. Quando um tratador solicita uma operação síncrona, a suspensão da co-rotina correspondente permite o retorno do controle ao *loop* de eventos, mantendo o estado do tratador. A recepção da tupla que contém o resultado da operação é associada a uma função de *callback* que reativa a co-rotina para que o tratador de eventos possa prosseguir sua execução. Caso seja necessário minimizar o custo de criação de tratadores de eventos, um *pool* de co-rotinas, como o descrito no Apêndice A, pode ser utilizado.

É importante observar que o uso de co-rotinas na implementação de aplicações orientadas a eventos não impõe um novo paradigma, ou estilo de programação. Nesse contexto, co-rotinas representam apenas uma facilidade conveniente para a implementação de tratadores de eventos que precisam ser decompostos, permitindo que o estado desses tratadores seja preservado e restaurado sem que seja necessário abrir mão de uma aparência de programação sequencial. O uso dessa facilidade é opcional, e não afeta a programação de tratadores de eventos que possam ser implementados por pequenos trechos de código ou por funções convencionais.

Um outro exemplo de combinação de processos, orientação a eventos e co-rotinas é a arquitetura proposta por Larus e Parkes [54] para a imple-

mentação de servidores em ambientes multi-processadores. Assim como na arquitetura SEDA, descrita na seção anterior, uma aplicação concorrente é decomposta em processos que representam uma sucessão de estágios. Cada estágio é responsável por um conjunto de operações assíncronas, e as solicitações e respostas a essas operações são enviadas através de mensagens. A distribuição de operações pelos diferentes estágios (denominada *cohort scheduling*) é feita de forma a agrupar computações que referenciam uma mesma região de código e dados, possibilitando ganhos de desempenho por executá-las em sequência e por reduzir a necessidade de sincronização. Dentro de cada estágio, a execução de operações é realizada de forma não preemptiva, porém uma operação pode ser suspensa — por exemplo, durante uma operação de entrada e saída — e retomada posteriormente. Os autores dessa arquitetura sugerem o uso de mecanismos como *fibers* para a manutenção do estado de operações suspensas. Contudo, não associam esse tipo de mecanismo ao conceito de co-rotinas, e o denominam de mecanismo de "continuações implícitas" (*implicit continuations*).

6.4 Co-rotinas versus threads

Neste capítulo, argumentamos que modelos de concorrência baseados em gerência cooperativa de tarefas e em orientação a eventos são alternativas vantajosas para o modelo de *multithreading*, permitindo o desenvolvimento de aplicações menos complexas, mais eficientes e muito mais fáceis de depurar. Discutimos também que em ambientes multi-processadores, a combinação de mecanismos de troca de mensagens com esses modelos pode substituir arquiteturas baseadas unicamente em *multithreading* com benefícios de maior simplicidade e melhor desempenho, apontando trabalhos que comprovam essa afirmação.

Podemos concluir, então, que para a maioria das aplicações concorrentes, seja em ambientes mono ou multi-processadores, o modelo de *multithreading* é inadequado, tanto com respeito à complexidade de programação quanto em relação a requisitos de correção e desempenho. À exceção de componentes de sistemas operacionais, a união de mecanismos de preempção e memória compartilhada pode ser conveniente apenas em cenários com requisitos rigorosos de justiça e responsividade, como aplicações de tempo real. Entretanto, a maioria das implementações de *multithreading* não oferece garantias reais com respeito a esses requisitos; em algumas implementações, como a de Java [56], a própria especificação do mecanismo de *multithreading*

não exige essas garantias. Dessa forma, mesmo no contexto de aplicações de tempo real, o uso de *multithreading* pode não oferecer vantagens.

Mostramos também neste capítulo que mecanismos de co-rotinas completas, além de permitir uma implementação trivial de gerência cooperativa de tarefas, provêem facilidades bastante convenientes para a implementação de aplicações orientadas a eventos. Podemos concluir, portanto, que uma linguagem que oferece co-rotinas completas não precisa oferecer *threads* ou qualquer outra construção adicional para prover um suporte básico adequado à programação concorrente independente de mecanismos do sistema operacional.