2
Escalonadores

Para executar um programa concorrente em máquinas mono-processadas, geralmente é necessário o auxílio de um escalonador, responsável por ditar qual tarefa deve ser executada e realizar todas as trocas de contexto necessárias para o correto funcionamento das tarefas [31]. Os escalonadores podem ser encontrados em diversos níveis do sistema, desde os núcleos dos sistemas operacionais modernos, controlando o acesso ao processador, até o nível das aplicações, com escalonadores para dividir a carga entre as diferentes atividades de uma aplicação. Neste capítulo vamos detalhar os modelos mais utilizados na implementação de escalonadores e discutir sua eficiência em diferentes níveis da execução de aplicações.

2.1
Escalonadores Preemptivos

O paralelismo nos permite usar os recursos disponíveis para computação de forma simultânea. Isso pode ocorrer em máquinas que dispõem de mais de um processador ou, na maioria dos casos, em sistemas onde existe a necessidade ou a oportunidade de realizar-se mais de uma tarefa ao mesmo tempo.

Em um sistema composto por máquinas multiprocessadas ou em arquiteturas distribuídas, as tarefas de uma aplicação podem ser executadas em processadores distintos, eliminando a necessidade de um escalonador. Em uma máquina mono-processada, o acesso aos recursos computacionais é bastante crítico para o desempenho do sistema e deve então ser controlado com cautela. É nesse cenário que introduzimos o conceito de *escalonadores preemptivos.*[3]

O escalonador é preemptivo quando é capaz de forçar a liberação do recurso computacional alocado ao processo que está executando. Em um escalonador preemptivo a preocupação com processos em *loop* ou bloqueados aguardando finalização de outros prodecimentos é minimizada, pois os

recursos em uso por esses processos podem sofrer preempção e darão lugar à execução de outros processos. Os principais modelos preemptivos utilizados para a execução de aplicações multitarefa são os modelos de processos e os modelos de *threads*.

2.1.1 Processos

Os primeiros sistemas computacionais não permitiam que mais de um programa compartilhasse o tempo de processamento disponível. Com a evolução dos sistemas veio a possibilidade de carregar múltiplos programas para a memória e executá-los concorrentemente. Dessa evolução surgiu a necessidade de um controle mais firme e de uma maior compartimentalização de diversos programas, resultando na noção de *processo* como um programa em execução [31].

Um programa pode ser encarado como uma série de instruções que são executadas seqüencialmente por um processador. Cada processo contém basicamente duas grandes áreas, a primeira, chamada de "tarefa", contém as instruções que devem ser executadas, uma porção da memória do sistema, onde o processo pode ler e escrever dados, e um conjunto de outros recursos alocados ao processo. A segunda, guarda as informações relativas à execução da tarefa, como um indicador da seqüência de processamento (*Program Counter*), que aponta para a próxima instrução a ser executada, um conjunto de registradores e o espaço de pilha local.

Quando se executam múltiplos processos, observa-se uma pseudo-paralelização da execução, conseguida com o rápido escalonamento desses processos em execução. Com blocos de memória e área de códigos distintas, quando um processo ganha acesso exclusivo ao processador, existe a necessidade de realizar a troca de contexto, processo que envolve restaurar o *program counter*, os registradores e definir o bloco de memória que será usado pelo novo processo [7]. Esse procedimento é levado em conta pelos escalonadores e seu custo influencia na freqüência com que essas trocas de contexto ocorrem.

A individualização das estruturas de armazenamento e execução de cada processo proporciona ao desenvolvedor um ambiente onde os dados escritos em memória por cada processo estão naturalmente protegidos [34]. Por outro lado, as tarefas de comunicação entre diferentes processos (IPC - *Interprocess Communication*) necessitam de estruturas como *sockets* ou *pipes* para executar a passagem de dados ou mensagens[32].

Essas tarefas de comunicação introduzem a necessidade de estruturas como semáforos e monitores, utilizadas para proteger regiões críticas em modelos de desenvolvimento multi-processo, porém trazem consigo um custo computacional associado, além de introduzir a possibilidade de situações onde um conjunto de diferentes linhas de execução não podem prosseguir pois estão aguardando algum evento que somente uma outra linha de execução desse conjunto pode produzir, causando o que chamamos de *deadlocks* [10, 34].

Um ambiente com múltiplos processos executando concorrentemente pode tornar o sistema mais eficiente pois permite que linhas de execução bloqueadas ao efetuar chamadas à dispositivos de entrada/saída ou ao sistema operacional dêem lugar a outra linha de execução não bloqueada, otimizando o uso do processador.

Porém, particularidades de desenvolvimento podem tornar o sistema complexo como a dificuldade natural em escrever códigos que utilizem o multiprocessamento de maneira eficiente; e mesmo desenvolvedores experientes mostram grande dificuldade no processo de depuração de tais códigos pela dificuldade associada na identificação da origem dos problemas.

O uso de múltiplos processos é bem adaptado para sistemas onde há pouca necessidade de sincronização entre seus componentes. Suas principais vantagens são o aumento da modularização do código, adquirido pela segregação de atividades independentes em processos distintos e o encapsulamento dado às operações bloqueantes normalmente associadas à entrada e saída de dados, que não precisam mais de tratamento especial e podem bloquear sem que o sistema como um todo fique bloqueado.

2.1.2 Threads

Os *threads*, conhecidos também como *lightweight processes* [31], podem ser vistos como as estruturas básicas de execução em uma CPU. O conceito de *thread* encapsula o *program counter*, os registradores e o espaço de pilha, que em conjunto com a "tarefa", forma o processo.

O modelo de *threads* adiciona a possibilidade de existirem múltiplas linhas de execução em um mesmo processo, criando um ambiente de pseudo-paralelismo similar ao conseguido com a execução de diversos processos em um mesmo processador; porém, com o compartilhamento do espaço de endereçamento e outros recursos alocados ao processo, a troca de contexto entre *threads* pode ser muito mais leve do que entre processos.

Não existe nenhum tipo de proteção para a área de memória compartilhada entre diferentes *threads*, o que permite a modificação dos dados contidos em variáveis globais do processo, facilitando as tarefas de comunicação, mas trazendo à tona uma série de problemas relacionados à coordenação da concorrência entre as linhas de execução [10].

Assim como no modelo de programação por processos, ainda observa-se a necessidade de estruturas controladoras de acesso as regiões críticas e se aplicam as mesmas dificuldades de desenvolvimento e depuração dos programas.

2.2 Escalonadores Colaborativos

O escalonador do sistema é colaborativo quando não possui meios de forçar a liberação dos recursos computacionais utilizados por um determinado processo. Desse modo, cada processo deve colaborar com os demais liberando espontaneamente os recursos alocados.

Essa estratégia elimina a necessidade de proteção nos dados compartilhados, porém é perigosa pois deixa a responsabilidade de liberar os recursos nas mãos do desenvolvedor de cada serviço. Um erro cometido no desenvolvimento do programa pode facilmente gerar um ciclo onde um único processo estará utilizando todo o tempo disponível para processamento.

Para fugir da complexidade tanto computacional quanto no desenvolvimento de código agregada pelo uso de múltiplos *threads*, observamos um crescente interesse em arquiteturas mais leves e mais simples. Em sistemas com alto grau de interação com o usuário ou orientados a eventos, é fundamental que o processamento das requisições ocorra de forma eficiente, permitindo que o sistema reaja à ação executada em pouco tempo.

Os escalonadores colaborativos no nível da aplicação raramente dependem de suporte muito específico do sistema operacional e por isso possuem um grau de portabilidade maior quando comparados aos modelos preemptivos. Em contrapartida, sua eficiência depende da existencia de suporte à operações não bloqueantes e sua implementação normalmente é feita um pouco abaixo do nível da aplicação, em *frameworks* projetados para encapsular o trabalho de escalonamento das tarefas e auxiliar o desenvolvimento de serviços. Nesta seção vamos descrever alguns modelos utilizados no escalonamento colaborativo de tarefas.

2.2.1 Reactor

O padrão de projeto *Reactor*[28] propõe a implementação de um demultiplexador de eventos síncrono, responsável por aguardar sincronamente o recebimento de novos eventos provenientes de uma ou mais fontes. Ao receber um evento, ele deve identificar um tratador específico da aplicação e para ele encaminhar o evento. Ao final do processamento do tratador associado, o controle do processador retorna para o *Reactor* que pode então receber novos eventos. Com esse modelo podemos ter uma separação dos métodos de demultiplexação e direcionamento dos eventos do código de tratamento desses, específico para cada aplicação.

Na implementação do *Reactor*, notamos um conjunto de componentes e estruturas que colaboram para a execução do modelo:

- **Descritor** - Para cada fonte de eventos, teremos um *Descritor* de identificação, normalmente fornecido pelo sistema operacional, e utilizado como chave para o encaminhamento do evento.
- **Demultiplexador de Eventos** - Componente que utiliza os *Descritores* para identificar o tratador de eventos a ser acionado. O demultiplexador de eventos pode ficar aguardando sinalizações do sistema em algum dos *Descritores* registrados e sinaliza quando o evento puder ser recuperado sem bloquear o sistema.
- **Tratadores de Eventos** - Definição da interface dos objetos que serão implementados na aplicação.
- **Tratadores Concretos de Eventos** - Implementações da interface padrão de tratamento de eventos. São estruturas dependentes da aplicação e responsáveis pelo tratamento dos eventos.
- **Reactor** - Formado pelo conjunto dos componentes acima, ele é responsável pela execução do ciclo de eventos da aplicação e do registro e remoção dos pares de *Descritores* e seus respectivos Tratadores.

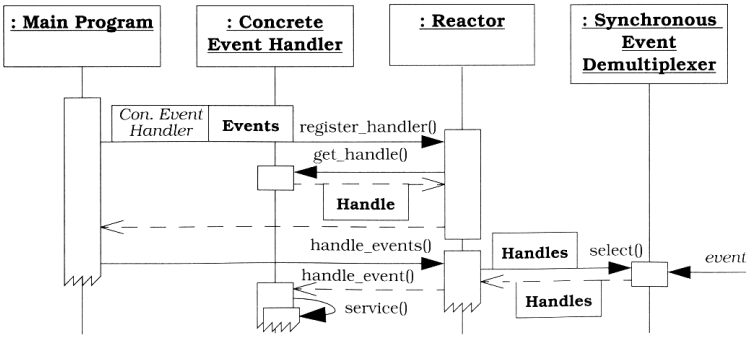

Figura 2.1: Diagrama de seqüência do *Reactor*

2.2.2 Proactor

O padrão de projeto *Proactor*[28] permite que aplicações orientadas a eventos coordenem a execução assíncrona das operações, aumentando o desempenho geral do sistema e possibilitando a execução de outros métodos entre a inicialização e o término das operações assíncronas, aproveitando melhor o tempo disponível da CPU.

O *Proactor* trabalha com um processador de operações assíncronas, em conjunto com uma fila de eventos de finalização, que associa os métodos iniciados aos seus respectivos finalizadores. Os seguintes participantes colaboram para formar o modelo *Proactor*:

- **Descritor** - Assim como no *Reactor*, o *Descritor* é fornecido pelo sistema operacional e é utilizado para identificar as operações ou uma fonte de origem de eventos de finalização.
- **Operação Assíncrona** - Usada na implementação do serviço, define operações do sistema que podem ser executadas assincronamente. Pode-se definir operações assíncronas como operações cujo processamento não faz com que a linha de execução fique bloqueada aguardando o resultado dessa operação, sendo executadas sem uma relação regular com outros eventos temporais do sistema.
- **Tratador de Finalização** - Interface dos objetos que serão usados no processamento dos resultados das operações assíncronas.
- **Tratador de Finalização Concreto** - Implementação dos métodos de processamento dos resultados, altamente dependente da aplicação.

– **Processador de Operações Assíncronas** - Responsável por executar as operações assíncronas e por interagir com a fila de eventos de finalização.

– **Fila de Eventos de Finalização** - Armazena os eventos até que sejam removidos pelo *demultiplexador de eventos*.

– **Demultiplexador de Eventos** - Aguarda que eventos sejam inseridos na fila, e é responsável por encaminhar a sinalização para o tratador correspondente.

– **Proactor** - Responsável pelo gerenciamento do demultiplexador e pela configuração dos demais componentes.

– **Iniciador** - Responsável por disparar as chamadas assíncronas.

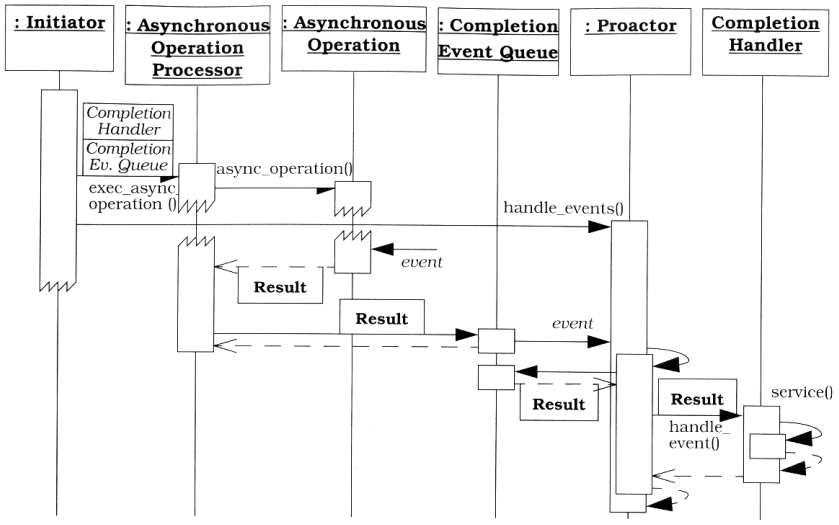

Figura 2.2: Diagrama de seqüência do *Proactor*

Na figura 2.2, observa-se uma aplicação iniciando uma operação assíncrona em um processador de operações, enviando os parâmetros necessários, bem como os tratadores de finalização concretos. Quando a aplicação está pronta para processar os eventos de finalização resultantes de suas operações assíncronas, é acionado o ciclo de tratamento de eventos do *Proactor*, que por sua vez ativa o demultiplexador que é responsável por aguardar os eventos de finalização e despachar os resultados da operação para o respectivo tratador concreto.

2.2.3 Corrotinas

Os primeiros registros do conceito de corrotinas surgiram em 1963 com a introdução do conceito de decomposição de um programa em processos seqüenciais que se comunicavam através de procedimentos de entrada e saída convencionais[9]. As corrotinas encontravam-se em um mesmo nível hierárquico, onde cada uma atuava como se fosse o programa principal. Esse modelo foi proposto para simplificar a cooperação entre os analisadores léxico e sintático de um compilador e seguia o modelo produtor/consumidor. Em 1967 a linguagem Simula [2] incorporou conceitos de corrotinas, porém com um modelo bastante complexo para que fosse usado em larga escala. Em seguida, outras linguagens incorporam o conceito de corrotinas como Modula-2[37] e Icon[13].

Desde então o modelo de corrotinas seguiu confuso e sem uma grande referência formal. Devido à falta de uniformidade na definição precisa de corrotinas, esta ainda permaneceu sem uso nas grandes linguagens de programação. Em 1980 Christopher Marlin [19], baseando-se no modelo apresentado na linguagem Simula em 1967, introduziu conceitos que até hoje são considerados base de referência para o modelo de corrotinas[21].

Durante toda a década de 90 houve um grande aumento na evidência de *multithreading* disponibilizado nas grandes linguagens, com a biblioteca *pthreads*[23, 29] em C e Java com suporte a *threads* em sua máquina virtual. Esse grande aumento do número de sistemas desenvolvidos sobre os ambientes multitarefa trouxe novamente o foco para soluções colaborativas como uma alternativa mais eficiente e menos suscetível a erros para os ambientes *multithread*. Ao mesmo tempo, o conceito de corrotinas também ganha força como um caso particular de *multithreading*, podendo ser encontrado como base para os *Fibers*, definidos como uma unidade de execução que deve ser escalonada pela própria aplicação e executa dentro do contexto do *thread* que os escalona [22]. Dentro do contexto de linguagens populares de *script* como Lua, as corrotinas são oferecidas como uma maneira mais portável e leve quando comparada aos *threads* tradicionais.

Atualmente ainda existem variações entre os modelos de corrotinas implementados. Essas diferenças representam um fator fundamental no poder de expressão e nas possibilidades de uso de corrotinas:

– **Mecanismo de transferência de controle** - Divide as implementações de corrotinas em “Simétricas” e “Assimétricas”. O primeiro grupo possui apenas um método para a transferência de controle, fa-

zendo com que todas as corrotinas executem em um mesmo nível hierárquico como mostrado na figura 2.3. O segundo grupo, das corrotinas assimétricas, mais conhecidas como semi-simétricas ou semi corrotinas, está ilustrado na figura 2.4 e trabalha com dois métodos: um para retomar (ou iniciar) a execução da rotina e outro para suspendê-la. Dessa forma, as corrotinas são executadas como rotinas subordinadas àquela que a invocou, em um modelo mais próximo das chamadas de rotinas tradicionais, tornando o modelo de transferência de controle das corrotinas assimétricas mais simples de compreender do que das corrotinas simétricas. É bastante comum que uma linguagem ofereça apenas um desses mecanismos de transferência de controle, visto que é simples mostrar que com o auxílio de um pequeno conjunto de funções é possível expressar um dos mecanismos em função do outro[21].

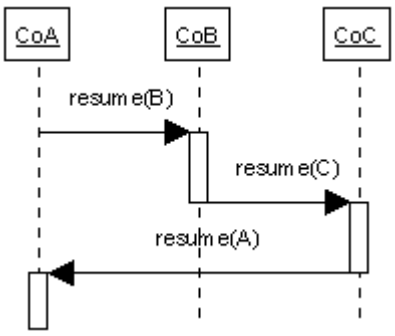

Figura 2.3: Transferência de controle em corrotinas simétricas.

- **Valores de primeira classe** - Um outro fator importante diz respeito ao fato das corrotinas serem ou não vistas pela linguagem como valores de primeira classe. Valores de primeira classe podem ser livremente manipulados pelo programador e podem ser invocados em qualquer lugar ou ordem, trazendo maior flexibilidade e mais possibilidades de uso para essas estruturas.
- ***Stackfulness*** - Corrotinas *stackful* são aquelas que permitem suspender sua execução mesmo quando dentro de algum método aninhado. Os modelos não-*stackful* são portando muito mais restritos no que diz respeito à aplicabilidade de corrotinas. A manutenção do contexto de execução em métodos aninhados é uma propriedade de extrema importância na implementação de ambientes multitarefa no nível da aplicação.

Pode-se então definir o conceito de corrotinas completas quando estas são oferecidas como valores de primeira classe e com objetos *stackful*, com

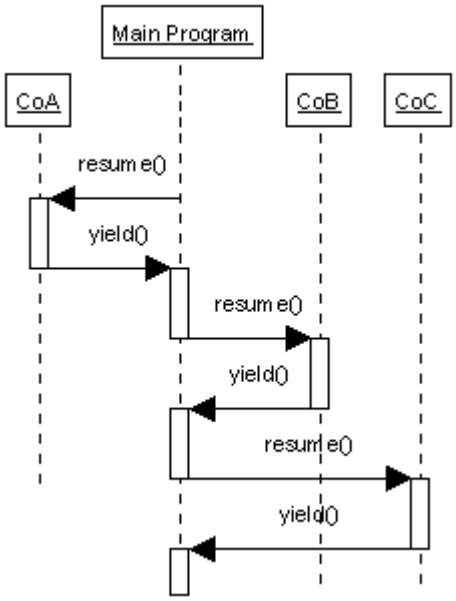

Figura 2.4: Transferência de controle em corrotinas assimétricas.

mecanismos simétricos ou assimétricos para a passagem de controle, visto que não há influência direta desses mecanismos sobre a expressividade das corrotinas. A linguagem Lua [27] implementa corrotinas completas e assimétricas e este será o modelo utilizado nesse trabalho para estudar o uso de escalonadores colaborativos no tratamento de eventos [20].

2.3 Considerações Finais

Neste trabalho é apresentado um modelo para o desenvolvimento de um escalonador não-preemptivo, com o objetivo de minimizar os problemas normalmente enfrentados durante o desenvolvimento de aplicações orientadas a eventos.

No campo dos sistemas distribuídos, observa-se que grande parte dos problemas encontrados surgem do gerenciamento de ações decorrentes do gerenciamento e processamento de eventos que usualmente são passados entre os pontos do sistema através de mensagens. A comunicação através de *sockets* bloqueantes, embora mais simples de gerenciar e com um fluxo lógico linear, faze com que toda a execução do programa fique bloqueada até a finalização da transmissão. Em um sistema onde existem rotinas com alto grau de independência, esse tempo bloqueado é perdido, diminuindo o desempenho total do sistema.

Quando existe uma aplicação distribuída com interação com o usuário, o tratamento dos eventos deve contemplar as mensagens recebidas de outros pontos de rede, e também as ações efetuadas atráves da interface com o usuário. O modelo apresentado deve prover mecanismos eficientes para que a integração de diferentes fontes de eventos possa ser feita de maneira transparente para o usuário final da aplicação.

O modelo proposto deve ter um alto grau de portabilidade, e deve ser de fácil integração em ambientes bastante heterogêneos. Essa limitação é imposta pela necessidade de execução em plataformas com severas restrições de recursos computacionais, como no caso de dispositivos móveis celulares, onde temos pouca memória disponível e nenhum suporte a preempção é oferecido pelo sistema.