2
Multithreading

O *multithreading* é um modelo amplamente utilizado para programação concorrente, caracterizado pela utilização de *threads* - também conhecidas como processos leves. As *threads* podem ser encaradas como versões simplificadas dos processos tradicionalmente utilizados em sistemas operacionais multitarefa. Na prática, um processo pode compreender uma única *thread*, o que representaria um único fluxo de execução, ou várias *threads*, o que representaria múltiplos fluxos de execução.

Um processo de um sistema operacional multitarefa precisa armazenar todas as informações de estado necessárias para permitir trocas de contexto, possui descritores de arquivos e espaço de endereçamento de memória exclusivos e, de maneira geral, é capaz de se comunicar com outros processos apenas através de mecanismos implementados pelo sistema operacional. Uma *thread*, por sua vez, possui apenas uma pilha e alguns registradores. As informações de estado, memória e descritores de arquivos são compartilhados com o processo que a contém. O compartilhamento de memória, em particular, pode ser utilizado como mecanismo para comunicação entre *threads* de um mesmo processo. As trocas de contexto entre *threads* são mais rápidas que as trocas de contexto entre processos, pois o ônus do processador ao carregar informações de estado é reduzido, face ao compartilhamento de recursos.

O termo *thread* normalmente é utilizado para designar *threads* no espaço do *kernel* do sistema operacional, ou seja, *threads* gerenciadas e escalonadas pelo *kernel*, também conhecidas como *kernel threads*. No entanto, é possível utilizá-lo também para designar *threads* no espaço do usuário, ou seja, *threads* gerenciadas e escalonadas por código executado no espaço do usuário, chamadas *user threads*. A ausência de especificação, portanto, denota referência a *kernel threads*.

O escalonamento das *threads* pode ser realizado de acordo com duas estratégias principais. No *multithreading preemptivo*, geralmente utilizado para o escalonamento de *kernel threads*, o sistema operacional aloca uma fração de tempo para a execução de cada *thread*, após a qual a execução da *thread* é interrompida e ocorre uma troca de contexto. No *multithreading cooperativo*

(ou *não-preemptivo*), o sistema operacional delega a gestão das trocas de contexto às próprias *threads*, ou seja, uma troca de contexto só ocorre quando é explicitamente requisitada por uma *thread*. No decorrer do trabalho, a ausência de especificação denota referência a *multithreading* preemptivo.

Ainda com relação ao escalonamento das *threads*, são comuns as referências a dois modelos distintos: o 1x1, ou *1-on-1*, e o MxN ou *M-on-N*. No modelo 1x1, cada *thread* representa uma entidade escalonável pelo sistema operacional. Esse modelo pode ser exemplificado pelas *kernel threads*. No modelo MxN, não existe uma equivalência entre *threads* e entidades escalonáveis pelo *kernel*. Esse modelo pode ser exemplificado por um sistema que utilize uma quantidade superior de *user threads*, escalonadas através de *kernel threads* em quantidade inferior.

Um dos principais usos do *multithreading* está na execução concorrente de múltiplos *threads* em computadores multiprocessados, com processadores multinúcleo ou com processadores com suporte a *hyper-threading*, o que permite a paralelização do processamento de instruções e, potencialmente, ganho de desempenho. O *multithreading* também pode ser utilizado para viabilizar um melhor aproveitamento do processador, ao possibilitar que operações bloqueantes resultem apenas no bloqueio das *threads* correspondentes, enquanto as demais *threads* continuam a ser executadas.

Entretanto, é justamente a execução concorrente, aliada ao compartilhamento de memória, que representa um dos maiores complicadores associados à utilização do *multithreading* preemptivo. A execução concomitante, por *threads* distintas, de operações de leitura ou escrita de dados armazenados na memória demanda a utilização criteriosa de mecanismos de sincronização.

Os mecanismos de sincronização baseados em memória compartilhada, em sua maioria, são voltados a promover exclusão mútua ou sincronização condicional (5). A exclusão mútua trata da atomicidade de operações, ou seqüências de operações, que envolvam dados em memória compartilhada. Seu objetivo é assegurar que determinados trechos de código somente sejam executados por uma *thread* de cada vez. Já a sincronização condicional trata da necessidade de retardar a execução de operações, ou seqüências de operações, até que alguma condição seja verdadeira.

A *espera ocupada*, os *semáforos* (12), as *regiões críticas condicionais* (18), os *guardas* (11) e os *monitores* (16, 19) estão entre os principais mecanismos desenvolvidos para sincronização com memória compartilhada. O funcionamento desses mecanismos envolve a utilização de operações bloqueantes, ou seja, operações que resultam no bloqueio de uma ou mais *threads*. Enquanto estão bloqueadas, as *threads* não podem prosseguir com a execução do código

até que ocorra alguma condição preestabelecida. O bloqueio de *threads* reduz o potencial de paralelismo e é particularmente nocivo para sistemas de tempo real ou para sistemas onde o alto desempenho é crucial.

Alguns estudos já foram desenvolvidos visando oferecer alternativas para superar essa limitação e reduzir os bloqueios (17, 34). As implementações de algoritmos para sincronização sem bloqueio são predominantemente baseadas em primitivas de hardware que permitem a execução atômica da seqüência ler (ou acessar), modificar e gravar.

A correta utilização dos mecanismos de sincronização baseados em memória compartilhada é difícil, e descuidos mínimos podem causar *deadlocks* ou inconsistências nos dados armazenados na memória. A preocupação com a sincronização, a dificuldade de depuração e o não-determinismo durante a execução tornam o processo de desenvolvimento de sistemas *multithreaded* reconhecidamente complexo (24).

No entanto, como apontado por Ousterhout (27), as críticas às *threads* não se limitam apenas às dificuldades de desenvolvimento. A obtenção de bom desempenho com *multithreading* preemptivo também não costuma ser trivial. A utilização pouco granular de mecanismos de sincronização tradicionais, como os mencionados anteriormente, pode resultar em concorrência reduzida, enquanto a utilização demasiadamente granular pode tornar o desenvolvimento ainda mais complexo e culminar com a redução do desempenho. A carência de bibliotecas capazes de assegurar o funcionamento correto de suas funções durante a execução simultânea por múltiplas *threads* - propriedade denominada *thread safety* - também é apontada como crítica ao *multithreading* preemptivo, uma vez que limita o desenvolvimento de sistemas que exploram esse modelo.

O *multithreading* preemptivo com memória compartilhada apresenta ainda diversas características que limitam sua escalabilidade. Essas características podem ser ilustradas com o auxílio de bibliotecas que implementam o padrão *POSIX Threads (Pthreads)* (20) em distribuições do sistema operacional Linux. A biblioteca LinuxThreads, adotada em *kernels* que precedem a versão 2.6, possui um vetor para armazenar as *threads* ativas que comporta no máximo 1.024 entradas. O *kernel* padrão, por sua vez, limita o número de processos ativos por usuário em 512 e considera as *threads* como processos para contabilizar esse limite. Como cada *thread* possui sua própria pilha, a criação de *threads* implica reserva de memória para conter as informações armazenadas nas respectivas pilhas. Ainda que o espaço seja apenas reservado, pois a alocação é dinâmica conforme a necessidade, o tamanho padrão da pilha (cerca de 2Mb) limita a quantidade de *threads* ativas.

A biblioteca Native POSIX Threads Library (NPTL), adotada em *kernels*

a partir da versão 2.6, flexibilizou algumas das limitações da LinuxThreads, como a tolerância da própria biblioteca a mais de 1.024 *threads* ativas, e foi responsável por inúmeras otimizações de desempenho. Contudo, ainda padece de algumas limitações, como o consumo de memória ocasionado pela reserva de espaço para as pilhas das *threads* (cerca de 8Mb[1] a 10Mb[2] por *thread*).

[1]Ubuntu Desktop Edition 7.10 - Gutsy Gibbon, NPTL 2.6.1.
[2]Red Hat Enterprise Linux AS release 4 (Nahant Update 6), NPTL 2.3.4.