Instituto de Matemática da UFRJ/NCE
Depto de Ciência da Computação
Mestrado em Informática
Lab. Sistemas Operacionais II
Professor Oswaldo Vernet

Multithread

Tema da monografia: Multithread
Autor: Luiz Paulo Maia
lpmaia@training.com.br
Data: 18/05/98

Multithread

Índice

1. Introdução

Tipicamente, os sistemas operacionais oferecem suporte a processos para o desenvolvimento de aplicações concorrentes. A utilização comercial de sistemas operacionais e aplicações multithread é recente, mas sua implementação está crescendo devido ao aumento de popularidade dos sistemas com múltiplos processadores, do modelo cliente-servidor e dos sistemas distribuídos.

Com threads, um processo pode ter diferentes partes do seu código sendo executadas concorrentemente ou simultaneamente, com muito menos overhead que utilizando múltiplos (sub)processos. Como os threads de um mesmo processo compartilham o mesmo espaço de endereço, a comunicação dos threads não envolve mecanismos lentos de intercomunicação entre processos.

Basicamente, multithreading é uma técnica de programação concorrente, que permite projetar e implementar aplicações paralelas de forma eficiente. O desenvolvimento de programas que exploram os benefícios da programação multithread não é simples. A presença do paralelismo introduz um novo conjunto de problemas, como a comunicação e sincronização de threads.

Existem diferentes modelos para a implementação de threads em um sistema operacional, onde desempenho, flexibilidade e custo devem ser avaliados atentamente. Além dos modelos tradicionais, com ênfase especial ao padrão Pthreads, apreciaremos arquiteturas novas e pouco difundidas.

Este trabalho pretende esclarecer o que vem a ser threads, sua utilização, vantagens, implementação e cuidados na programação concorrente. Antes porém, devemos apresentar o conceito de processo, que está intimamente relacionado a threads.

2. Processos

O conceito de processo surgiu nos anos 60, sendo a base da multiprogramação e dos sistemas de tempo compartilhado (time-sharing). O processo pode ser entendido como um programa em execução, só que seu conceito é mais abrangente. Este conceito torna-se mais claro quando pensamos de que forma os sistemas multiprogramáveis (multitarefa) atendem os diversos usuários (tarefas) e mantêm informações a respeito dos vários programas que estão sendo executados concorrentemente.

Como sabemos, um sistema multiprogramável simula um ambiente de monoprogramação para cada usuário, isto é, cada usuário do sistema tem a impressão de possuir o processador exclusivamente para ele. Nesses sistemas, o processador executa a tarefa de um usuário durante um intervalo de tempo (time-slice) e, no instante seguinte, está processando outra tarefa. A cada troca, é necessário que o sistema preserve todas as informações da tarefa que foi interrompida, para quando voltar a ser executada não lhe faltar nenhuma informação para a continuação do processamento. A estrutura responsável pela manutenção de todas as informações necessárias à execução de um programa, como conteúdo de registradores e espaço de memória, chama-se *processo*.

O conceito de *processo* pode ser definido como sendo o ambiente onde se executa um programa. Um mesmo programa pode produzir resultados diferentes, em função do processo no qual ele é executado. Por exemplo, se um programa necessitar abrir cinco arquivos simultaneamente, e o processo onde será executado só permitir que se abram quatro, o programa será interrompido durante a sua execução.

O processo pode ser dividido em três elementos básicos: contexto de hardware, contexto de software e espaço de endereçamento, que juntos mantêm todas as informações necessárias à execução do programa (Fig. 1).

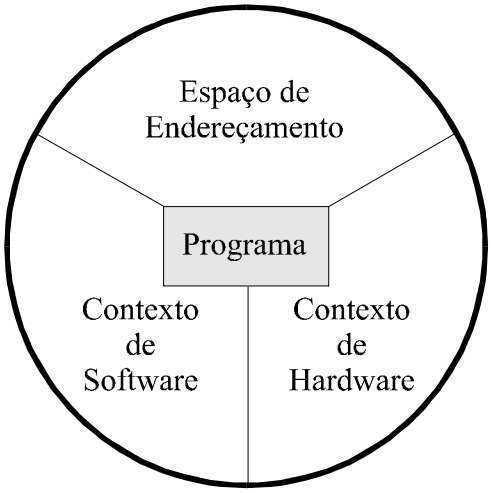

Fig. 1 Processo

Contexto de hardware constitui-se, basicamente, no conteúdo de registradores: program counter (PC), stack pointer (SP) e bits de estado. Quando um processo está em execução, o contexto de hardware está armazenado nos registradores do processador. No momento em que o processo perde a utilização da UCP, o sistema salva suas informações.

Contexto de software especifica características do processo que vão influir na execução de um programa, como, por exemplo, o número máximo de arquivos abertos simultaneamente ou o tamanho do buffer para operações de E/S. Essas características são determinadas no momento da criação do processo, mas algumas podem ser alteradas durante sua existência. O contexto de software define basicamente três grupos de informações de um processo: sua identificação, suas quotas e seus privilégios.

Espaço de endereçamento é a área de memória do processo onde o programa será executado, além do espaço para os dados utilizados por ele. Cada processo possui seu próprio espaço de endereçamento, que deve ser protegido do acesso dos demais processos.

3. Threads

Até o final da década de 70, os sistemas operacionais mais populares, como Tops-10 (DEC), MVS (IBM) e Unix (Bell), suportavam apenas processos, sem threads. Em 1979, durante, o desenvolvimento do sistema operacional Toth, foi introduzido o conceito de processos lightweight (peso leve), onde o espaço de endereçamento de um processo poderia ser compartilhado por vários programas.

Apesar do conceito revolucionário, a idéia não foi aceita comercialmente e somente em meados de 1980, com o desenvolvimento do sistema operacional Mach, na Universidade de Carnegie Mellon, ficou clara a separação entre o conceito de processo e threads. Atualmente, o mesmo conceito pode ser encontrado em sistemas operacionais, como OS/2 (IBM), Solaris (Sun) e Windows NT (Microsoft), dentre outros.

3.1 Ambiente Monothread

Um programa (tarefa) é uma seqüência de instruções, composto por desvios, repetições (iterações) e chamadas de procedimentos e/ou funções. Em um ambiente de programação monothread, um processo suporta apenas um programa no seu espaço de endereçamento e apenas uma instrução do programa é executada. Caso seja necessário criar-se aplicações concorrentes e paralelas são implementados múltiplos processos independentes e/ou subprocessos.

A utilização de (sub)processos independentes permite dividir uma aplicação em partes que podem trabalhar de forma concorrente. Por exemplo, suponha que um processo seja responsável pelo acesso a um banco de dados e existam vários usuários solicitando consultas sobre esta base. Caso um usuário solicite um relatório impresso de todos os registros, os

demais usuários terão de aguardar até que a operação termine. Com o uso de múltiplos processos, cada solicitação implicaria a criação de um novo processo para atendê-la, aumentando o throughput da aplicação. Existem, porém, dois problemas neste tipo de abordagem.

O uso de (sub)processos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado, o sistema deve alocar recursos (contexto de hardware, contexto de software e espaço de endereçamento) para cada processo, além de consumir tempo de UCP neste trabalho. No caso do término do processo, o sistema dispensa tempo para desalocar recursos previamente alocados. Na Fig. 2 existem três processos, cada um com seu próprio contexto de hardware, contexto de software e espaço de endereçamento.

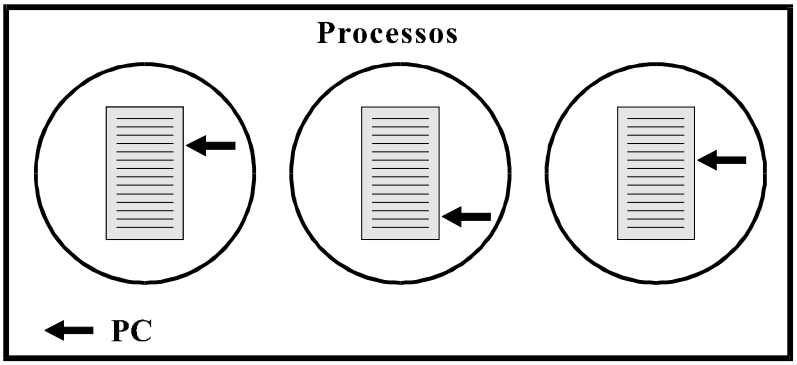

Fig. 2 Ambiente monothread

Como cada processo possui seu próprio espaço de endereçamento, a comunicação entre os (sub)processos torna-se difícil e lenta, pois utiliza mecanismos tradicionais como pipes, sinais, semáforos, memória compartilhada ou troca de mensagem. Além disto, o compartilhamento de recursos comuns aos (sub)processos concorrentes, como memória e arquivos abertos, não é simples.

3.2 Ambiente Multithread

Na tentativa de diminuir o tempo gasto na criação/eliminação de (sub)processos, bem como economizar recursos do sistema como um todo, foi introduzido o conceito de thread. Em um ambiente de múltiplos threads (multithread), não é necessário haver vários processos para se implementar aplicações concorrentes. No ambiente multithread, cada processo pode responder a várias solicitações concorrentemente ou mesmo simultaneamente, se houver mais de um processador. Na Fig. 3 existe apenas um processo com três threads de execução, cada um com seu program counter (PC).

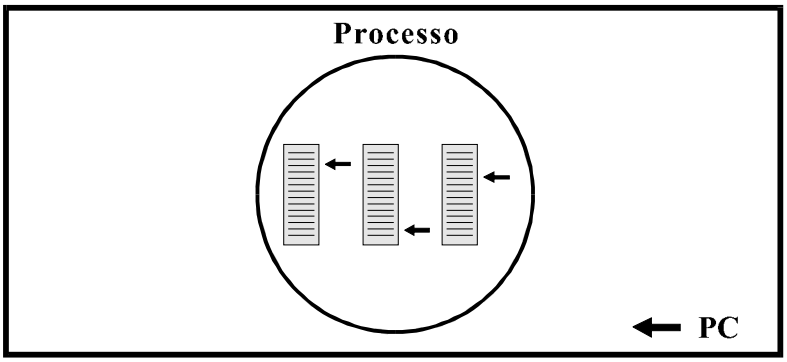

Fig. 3 Ambiente Multithread

Em um ambiente multithread, não existe a idéia de um programa, mas de threads (linhas). O processo, neste ambiente, tem pelo menos um thread de execução, podendo compartilhar o seu espaço de endereçamento com inúmeros threads, que podem ser executados de forma concorrente e/ou simultânea, no caso de múltiplos processadores.

Threads compartilham o processador da mesma maneira que um processo. Por exemplo, enquanto um thread espera por uma operação de E/S, outro thread pode ser executado. Cada thread possui seu próprio conjunto de registradores (contexto de hardware), porém compartilha o mesmo espaço de endereçamento com os demais threads do processo. Os threads de um mesmo processo compartilham, além do espaço de endereçamento, outros atributos, como temporizadores e arquivos, de forma natural e eficiente (Fig. 4).

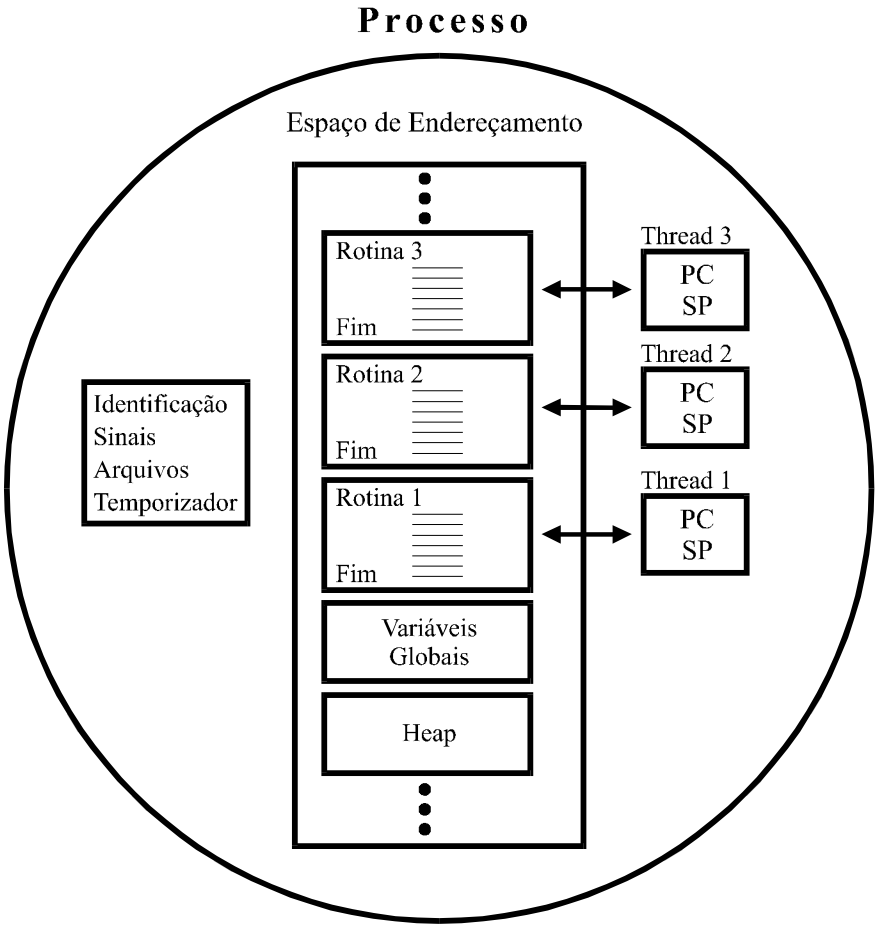

Fig. 4 Compartilhamento de recursos

Quando um thread está sendo executado, o contexto de hardware está armazenado nos registradores do processador. No momento em que um thread perde a utilização do processador, o sistema salva suas informações. Threads passam pelos mesmos estados que um processo, ou seja, execução, espera e pronto.

A grande diferença entre subprocessos e threads é em relação ao espaço de endereçamento. Enquanto subprocessos possuem, cada um, espaços independentes e protegidos, threads compartilham o mesmo espaço de endereçamento do processo, sem nenhuma proteção, permitindo que um thread possa alterar dados de outro thread. Apesar dessa possibilidade, threads são desenvolvidos para trabalhar de forma cooperativa, voltados para desempenhar uma tarefa em conjunto.

3.3 Vantagens

Programas concorrentes que utilizam múltiplos threads são mais rápidos do que implementados como múltiplos (sub)processos. Como os threads compartilham os recursos do processo, as operações de criação, troca de contexto e eliminação dos threads geram um ganho de desempenho.

Como todos os threads em um processo compartilham o mesmo espaço de endereçamento, a comunicação entre os threads pode ser feita utilizando o compartilhamento de memória (shared memory) de forma rápida e eficiente. De forma

semelhante ao compartilhamento de memória, os threads dentro do mesmo processo podem compartilhar facilmente outros recursos, como descritores de arquivos, temporizadores (timers), sinais, atributos de segurança etc.

A utilização dos recursos de hardware, como UCP, discos e periféricos, pode ser feita de forma concorrente pelos diversos threads, significando uma melhor utilização dos recursos computacionais disponíveis.

Em algumas aplicações, a utilização de threads pode melhorar o desempenho da aplicação apenas executando tarefas secundárias (background), enquanto operações, como entrada e saída, estão sendo processadas. Aplicações como editores de texto, planilhas, aplicativos gráficos e processadores de imagens são especialmente beneficiados quando desenvolvidos com base em threads.

As vantagens oferecidas em ambientes com múltiplos processadores podem ser aproveitadas em aplicações multithread, dispensando qualquer preocupação por parte dos desenvolvedores. O escalonamento é realizado com base na prioridade de cada thread, sendo selecionado aquele com maior prioridade. Dessa forma, o programador pode determinar níveis de prioridade das threads em função da importância de cada uma. Aplicações que podem ser naturalmente paralelizadas, como ordenação e pesquisa, obtém ganhos de desempenho por serem executadas em múltiplos processadores simultaneamente.

Em ambientes distribuídos, threads são essenciais para solicitações de serviços remotos. Em um ambiente monothread, se uma aplicação solicita um serviço remoto, ela pode ficar esperando indefinidamente, enquanto aguarda pelo resultado. Em um ambiente multithread, um thread pode solicitar o serviço remoto, enquanto a aplicação pode continuar realizando outras atividades úteis. Já para o processo que atende a solicitação, múltiplos threads permitem que diversos pedidos sejam atendidos concorrentemente e/ou simultaneamente.

Não apenas aplicações tradicionais podem fazer uso dos benefícios do multithreading. O núcleo do sistema operacional também. Quando o kernel é multithread é possível atender a várias solicitações, como, por exemplo, em device drivers. Neste caso, mecanismos de sincronização devem ser utilizados para proteger as estruturas de dados internas do núcleo.

4. Arquitetura e Implementação

O conjunto de chamadas (primitivas) disponíveis para que uma aplicação utilize as facilidades dos threads é chamado de pacote de threads (threads package). Existem diferentes abordagens na implementação deste pacote em um sistema operacional, o que influenciará o desempenho, concorrência e modularidade das aplicações multithread.

Threads podem ser oferecidos pelo próprio núcleo do sistema operacional (thread em modo kernel), por uma biblioteca de rotinas fora do núcleo do sistema (thread em modo usuário), uma combinação de ambos (híbrido), seguir o novo modelo de scheduler activations ou a novíssima arquitetura de processadores multithread. A tabela abaixo sumariza as diversas arquiteturas para diferentes sistemas operacionais.

Sistemas	**Arquitetura**
Distributed Computing Environment (DCE)	Modo usuário
Xerox's Portable Common Runtime (PCR)	Modo usuário
DEC OpenVMS versão 6.*	Modo usuário
Windows NT	Modo kernel
Digital Unix	Modo kernel
DEC OpenVMS versão 7.*	Modo kernel
IBM OS/2	Modo kernel
Sun Solaris versão 2.*	Modo híbrido
University of Washington FastThreads	Scheduler activations
Tera MTA	Processador multithread

4.1 Threads em Modo Usuário

Threads em modo usuário são implementas por chamadas a uma biblioteca de rotinas que são linkadas e carregadas em tempo de execução (run-time) no mesmo espaço de endereçamento do processo e executadas em modo usuário (fig. 5). O

sistema operacional não sabe da existência de múltiplos threads, sendo responsabilidade da biblioteca gerenciar e sincronizar os diversos threads existentes.

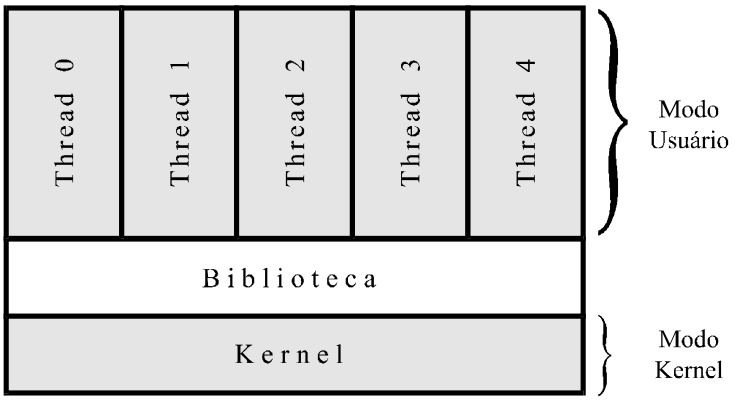

Fig. 5 Threads em modo usuário

A primeira vantagem deste modelo é a possibilidade de sistema operacional que não suporta threads, implementar aplicações multithreads. Utilizando a biblioteca, múltiplos threads poder ser utilizados, compartilhando o mesmo espaço de endereçamento do processo e outros recursos.

Threads em modo usuário são rápidos e eficientes, por dispensar acesso ao kernel do sistema para a criação, eliminação, sincronização e troca de contexto das threads. A biblioteca oferece todo o suporte necessário em modo usuário, sem a necessidade de chamadas ao sistema (system calls).

O sistema operacional desconhece a existência dos threads, sendo responsabilidade da biblioteca particionar o tempo de UCP do processo (time-slice) entre os diversos threads existentes. Como cada aplicação possui sua cópia da biblioteca, é possível implementar uma política de escalonamento diferente, em função da sua necessidade.

Apesar das vantagens, threads em modo usuários são difíceis de implementar. Um grande problema surge quando um thread utiliza uma chamada ao sistema que a coloca em estado de espera (chamada bloqueante ou síncrona), como, por exemplo, uma operação de entrada/saída. Neste caso, como o sistema operacional encara o processo com apenas um thread, o sistema coloca todo o processo em estado de espera, mesmo havendo outros threads prontos para a execução.

Para contornar o problema das chamadas síncronas, a biblioteca tem que implementar todas as chamadas que possam causar o bloqueio de um thread. Sempre que uma chamada ao sistema é realizada, a biblioteca verifica a possibilidade de bloqueios e intervém, utilizando uma chamada assíncrona própria. Todo este controle é transparente para o usuário e para o sistema operacional.

Outro problema do pacote em modo usuário está no compartilhamento de variáveis da biblioteca multithread sem a devida sincronização. Bibliotecas que apresentam este tipo de problema são ditas não reentrante ou thread-safe (segura). Para contorna-lo, a biblioteca deve fornecer para cada thread um conjunto individual de variáveis ou o programador deve intervir para a garantir a integridade das variáveis. Não apenas a biblioteca multithread deve ser thread-safe, mas também todas as bibliotecas oferecidas pelo sistema operacional utilizadas pela aplicação.

Talvez um dos maiores problemas na implementação de threads em modo usuário, seja o tratamento de sinais por cada thread individualmente. Como o sistema reconhece apenas processos e não threads, os sinais enviados para um processo devem ser reconhecidos pela biblioteca e encaminhado ao thread correspondente. Um problema crítico no tratamento de sinais é o recebimento de interrupções de clock, fundamental para a implementação do time-slice. Logo, para oferecer escalonamento preemptivo, a biblioteca deve receber sinais de temporização, interromper o thread em execução e realizar a troca de contexto.

Outro grande problema relacionado ao escalonamento é a incapacidade de múltiplos threads em um processo serem executados por processadores diferentes simultaneamente, em ambientes com múltiplos processadores. Esta restrição limita drasticamente o grau de paralelismo da aplicação, já que os threads de um mesmo processo podem ser executados em somente um processador de cada vez.

4.2 Threads em Modo Kernel

Threads em modo kernel são implementadas diretamente pelo núcleo do sistema, por chamadas ao sistema (system calls) que oferecem todas as funções de gerenciamento e sincronização (Fig. 6). O sistema operacional (escalonador) sabe da existência de cada thread e pode escalona-los individualmente. No caso de múltiplos processadores, os threads de um mesmo processo podem ser executados simultaneamente.

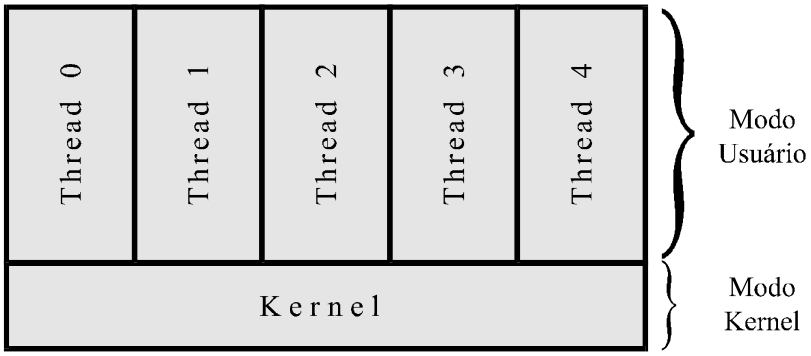

Fig. 6 Threads em modo kernel

Os problemas apresentados para a implementação de pacotes em modo usuário não são encontrados neste modelo, como compartilhamento de variáveis, tratamento de sinais, chamadas síncronas, escalonamento em múltiplos processadores etc.

O grande problema para pacotes em modo kernel é o seu desempenho, sendo da ordem de dez vezes mais lento que o modo usuário. Enquanto que pacotes em modo usuário todo tratamento é feito sem a ajuda do sistema operacional, ou seja, sem a mudança do modo de acesso (usuário-kernel-usuário), pacotes em modo kernel utilizam chamadas ao sistema e conseqüente mudança de modo de acesso.

4.3 Threads em Modo Híbrido

Nesta arquitetura existe a idéia de combinar as vantagens de threads implementados em modo usuário e modo kernel. Para facilitar a explicação deste modelo, chamaremos os threads em modo kernel de TMKs e os de modo usuário de TMUs.

Um processo pode ter vários TMKs e, por sua vez, um TMK pode ter vários TMUs. O núcleo do sistema reconhece os TMKs e pode escaloná-los individualmente. Um TMU pode ser executado em um TMK, em um determinado momento, e no instante seguinte ser executado em outro.

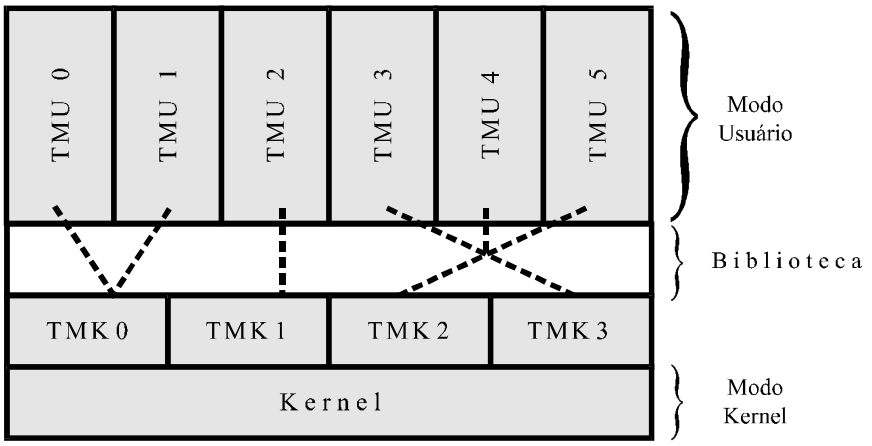

Fig. 7 Threads em modo híbrido

O programador desenvolve a aplicação em termos de TMUs e especifica quantos TMKs estão associados ao processo. Os TMU são mapeados em TMK enquanto o processo está sendo executado. O programador pode utilizar apenas TMKs, TMUs ou uma combinação de ambos, como mostra a Fig. 7.

O pacote híbrido, apesar da maior flexibilidade, também apresenta problemas herdados de ambas as implementações. Por exemplo, quando um TMK realiza uma chamada bloqueante, todos os TMUs são colocados no estado de espera. TMUs que desejam utilizar vários processadores devem utilizar diferentes TMKs, o que influenciará no desempenho.

4.4 Scheduler Activations

Os problemas apresentados no pacote de threads híbrido existem devido a falta de comunicação entre os threads em modo usuário e modo kernel. O modelo ideal deveria utilizar as facilidades do pacote em modo kernel com o desempenho e flexibilidade do modo usuário.

Introduzido no início da década de 90, na Universidade de Washington, este pacote combina o melhor do dois mundos, mas ao contrário de multiplexar os threads em modo usuário entre os de modo kernel, o núcleo do sistema troca informações com a biblioteca de threads utilizando uma estrutura de dados chamada scheduler activation (Fig. 8).

A maneira de alcançar um melhor desempenho é evitar a mudanças de acessos (usuário-kernel-usuário) desnecessárias. Caso um thread utilize uma chamada ao sistema que o coloque no estado de espera, não é necessário que o kernel seja ativado. Basta que a própria biblioteca em modo usuário possa escalonar outro thread. Isto é possível porque a biblioteca em modo usuário e o kernel se comunicam e trabalham de forma cooperativa. Cada camada implementa seu escalonamento de forma independente, porém trocando informações quando necessário.

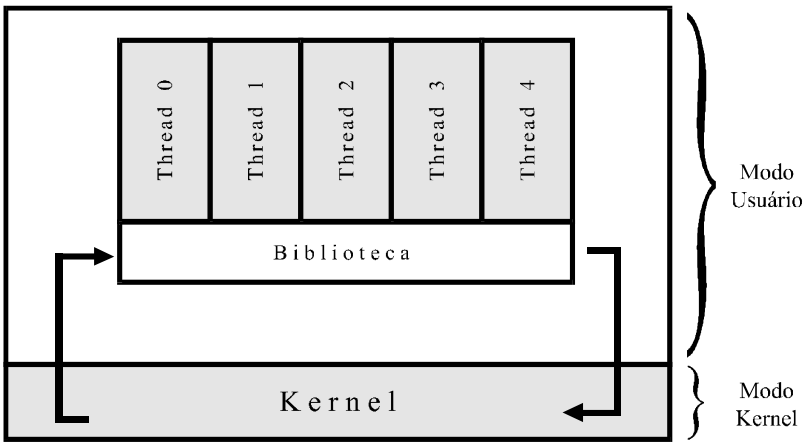

Fig. 8 Scheduler activations

4.5 Processadores Multithread

O conceito de thread reduziu significativamente o tempo gasto na troca de contexto entre processos, mas para algumas aplicações este tempo continua alto. Para reduzir o tempo de troca e, assim, aumentar a concorrência, uma nova arquitetura de processadores está sendo oferecida. Em uma arquitetura de processador multithread, a troca de contexto dos threads é feita por hardware, diretamente no processador, quase eliminado a latência da troca de contexto.

Este tipo de arquitetura começou a ser estudada na década de 50, mas foi apenas na década de 70 que o primeiro sistema multithread foi comercializado, o Heterogeneous Element Processor (HEP) da Denelcor. Atualmente, o MTA da Tera Computer vem sendo comercializado com base nesta nova tecnologia.

Em uma arquitetura multithread, os threads são mapeados em contextos de hardware, que incluem registradores, bits de estado e contador de intruções (PC). Cada contexto representa um thread que pode ser executado, aguardando apenas uma chance de ser executado. Por limitações de hardware, nem todos os threads estarão mapeados em um contexto de hardware.

Apesar da grande melhoria de desempenho, arquiteturas multithread são mais caras e complexas que as arquiteturas tradicionais de alto desempenho.

5. Pthreads

Uma das grandes dificuldades da utilização de threads em aplicações em geral foi a ausência de um padrão. Em 1995, o padrão POSIX (Portable Operating System Interface) P1003.1c foi aprovado. Com este padrão, também conhecido como Pthreads, aplicações comerciais multithread deverão tornar-se mais comuns. A tabela abaixo apresenta as principais APIs definidas pelo padrão POSIX.1c.

Descrição	**POSIX Thread API**
Gerenciamento	Pthread_create Pthread_exit pthread_join pthread_kill Pthread_t pthread_self
Exclusão mútua	Pthread_mutex_init Pthread_mutex_trylock Pthread_mutex_lock Pthread_mutex_unlock Pthread_mutex_destroy
Variáveis condicionais	Pthread_cond_init Pthread_cond_destroy Pthread_cond_wait Pthread_cond_timedwait Pthread_cond_signal Pthread_cond_broadcast

O padrão Pthreads é largamente encontrado em ambientes Unix, geralmente implementado em aplicações escritas em Linguagem C.

5.1 Arquitetura

O POSIX.1c pode ser implementado apenas em modo usuário, modo kernel ou uma combinação de ambos (híbrido). O padrão utiliza a abordagem de orientação a objetos para representar suas propriedades, como tamanho de pilha, política de escalonamento e prioridades para os threads. Diversos threads podem ser associados ao mesmo objeto e, caso uma alteração seja feita, todos os threads associados serão afetados.

No padrão POSIX.1c, threads são criados e eliminados dinamicamente, conforme a necessidade. Além disto, o padrão oferece mecanismos de sincronização, como semáforos, mutexes e variáveis condicionais.

No ambiente Pthreads, sinais são enviados para o processo, não threads. Para cada thread existe uma máscara de sinais e, quando chega um sinal, o processo deve identificar para qual thread se destina. O tratamento de sinais é muito dependente da implementação do pacote em um determinado sistema operacional.

O POSIX threads exige que todas as bibliotecas e chamadas ao sistema sejam thread-safe, ou seja, se um fabricante deseja oferecer o pacote, deverá reescrever grande parte do sistema para torná-lo compatível.

5.2 Sincronização

O Pthreads oferece inúmeras funções para a sincronização entre threads. A seguir descrevemos, resumidamente, estes mecanismos:

- Mutexes

Uma variável mutex (Mutual Exclusion) permite implementar a exclusão mútua, ou seja, o acesso a uma região crítica é feito de forma a impedir os problemas de sincronização (race conditions).

- Variáveis condicionais

Uma variável condicional permite associar uma variável a um evento, que poder ser, por exemplo, um contador alcançando um determinado valor ou um flag sendo ligado/desligado.

Em uma aplicação multithread, a utilização de variáveis condicionais permite criar um mecanismo de comunicação entre os diversos threads, que, por sua vez, permite a execução condicional em torno de eventos (sincronização).

- Semáforos

Semáforos são semelhantes aos mutexes, com a diferença que, enquanto um mutex pode variar seu valor entre zero e um, semáforos podem assumir outros valores, permitindo sua utilização como contadores.

Semáforos estão disponíveis apenas em sistemas que oferecem suporte ao padrão POSIX com extensões para tempo-real (POSIX.1b).

6. Modelos de Programação

O desenvolvimento de aplicações multithread não é simples, pois exige que a comunicação e o compartilhamento de recursos entre os diversos threads seja feito de forma sincronizada, para evitar problemas de inconsistências e deadlock. Além das dificuldades naturais no desenvolvimento de aplicações paralelas, existe também o problema de sua depuração.

Um fator importante em aplicações multithread é o número de threads da aplicação e como são criados e eliminados. Se uma aplicação cria um número excessivo de threads poderá ocorrer um overhead no sistema, gerando uma queda de desempenho.

Dependendo da implementação a definição do número de threads pode ser dinâmica ou estática. Quando a criação/eliminação dos threads é dinâmica, estes são criados/eliminados conforme a demanda da aplicação, oferecendo uma grande flexibilidade. Já em ambientes estáticos, o número de threads é definido na criação do processo onde a aplicação será executada, limitando o seu throughput.

As aplicações que obtém ganhos sendo desenvolvidas como múltiplos threads devem obrigatoriamente poder ser divididas em módulos independes, que possam ser executadas concorrentemente e/ou simultaneamente. Se uma aplicação realiza várias operações de E/S e/ou trata eventos assíncronas, como acesso à rede ou interrupções de hardware ou software, a programação multithread pode aumentar seu desempenho e throughput, mesmo em ambientes com um único processador. Aplicações como servidores de banco de dados, servidores de arquivo e impressão são ideais para multithreading, pois lidam com inúmeras operações de E/S e eventos assíncronos.

Aplicações tipicamente orientadas ao processador (CPU-bound) e processadas em ambientes com múltiplos processadores, também podem obter melhorias de desempenho. Aplicações que trabalham com matrizes, equações, ordenações, pesquisas e processamento de imagens podem obter ganhos significativos com o aumento de paralelismo é permitido.

Existem maneiras diferentes de dividir e organizar uma aplicação multithread. Dependendo da natureza do problema, podemos escolher basicamente entre três modelos de programação.

6.1 Modelo de Grupo de Trabalha

No modelo de grupo de trabalho (workgroup, peer-to-peer ou workcrew), todos os threads trabalham de forma concorrente para a solução cooperativa do problema, onde cada thread executa uma tarefa bem específica.

Neste modelo, um thread é responsável por criar todos os demais threads da aplicação, na criação do processo. Pode ocorrer de um thread não poder atender a uma solicitação, por já estar ocupado. Para contornar esta limitação, deve-se manter uma fila de terefas pendentes por thread.

O modelo de grupo de trabalho é adequado para aplicações que têm um número conhecido de tarefas a serem executadas. Por exemplo, suponha um editor de textos com múltiplos threads, onde um thread é responsável pela entrada de dados pelo teclado, um segundo pela a exibição no monitor e um terceiro pela gravação periódica do texto em um arquivo em disco (Fig. 9).

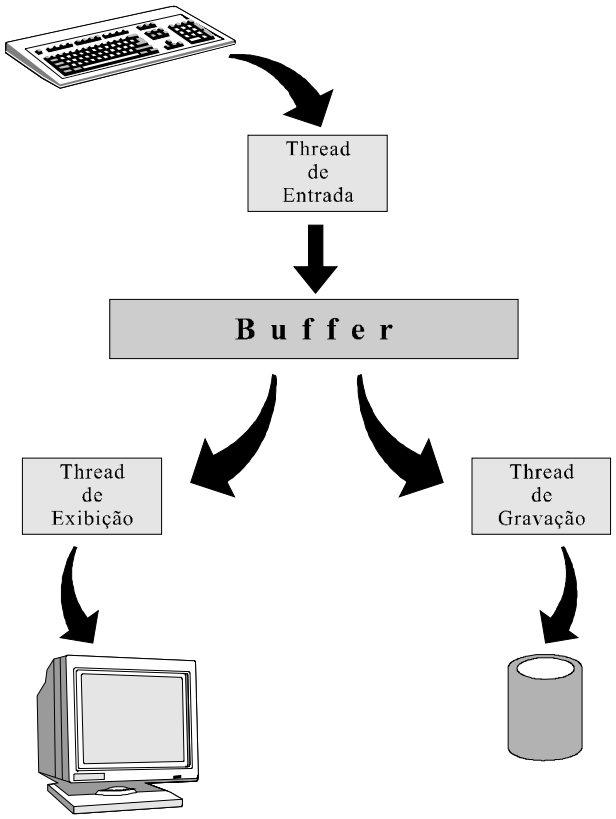

Fig. 9 Modelo de grupo de trabalho

Um outro exemplo muito comum do uso de threads pode ser apreciado em um simples programa que realiza uma cópia de um arquivo e permite o cancelamento da operação. No caso de não utilizarmos a solução multithread, o programa periodicamente deve testar se a opção de cancelamento foi acionada (polling). Já na solução por thread, podemos imaginar que existe um thread responsável pela cópia e um outro preocupado apenas com o cancelamento da operação.

6.2 Modelo Mestre-escravo

No modelo mestre-escravo (master-slave) ou chefe-trabalhador (boss-worker) existe um thread principal (mestre), responsável por criar, eliminar e coordenar os threads que efetivamente executarão as tarefas (escravos).

O mestre pode criar os threads escravos dinamicamente a cada solicitação ou cria-los previamente (thread-pool) na inicialização do processo. A segunda opção permite alcançar um melhor desempenho, na medida em que elimina o tempo gasto na criação/eliminação dos threads.

Este modelo é indicado para aplicações que devem manipular várias solicitações concorrentes, como, por exemplo, uma aplicação cliente-servidor. Neste tipo de aplicação, o servidor deve atender a múltiplas solicitações dos clientes. Neste caso, cada solicitação de um cliente pode ser tratada por uma thread independente criada pelo servidor. Neste modelo, o número de threads é variável, dependendo da carga que a aplicação está sofrendo (Fig. 10).

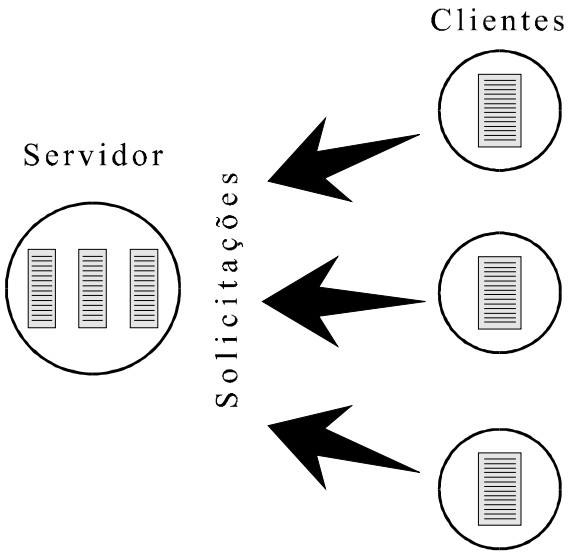

Fig. 10 Modelo mestre-escravo

6.3 Modelo de Pipeline

O modelo de pipeline deve ser empregado quando um problema pode ser dividido em vários threads, por onde os dados ao sair de um thread podem ser processados pelo thread seguinte, como em uma linha de produção.

Um exemplo deste modelo pode ser encontrado em linguagens de controle, onde a saída de um comando é a entrada para um próximo, que por sua vez gera a entrada para um outro e, assim, de forma sucessiva, como no redirecionamento de entrada/saída. Outro exemplo de aplicação onde o pipeline pode ser útil pode ser apreciado em aplicações para processamento de imagens.

Referências

[Byrd_1] Gregory T. Byrd & Mark A. Holliday. *Multithreaded Processor Architectures*. IEEE Spectrum, August, 1995.

[Kleiman_1] Steve Kleiman, Devang Shah & Bart Smalders. *Programming with Threads*. SunSoft Press, Prentice-Hall, 1996.

[Nichols_1] Bradford Nichols, Dick Buttar & Jacqueline Firrell. *Pthreads Programming*. O'Reilly Press, 1996.

[Pham_1] Thuan Q. Pham & Pankaj K. Gang. *Multithreaded Programming with Windows NT*, Prentice-Hall, 1996

[Robbins_1] Kay A. Robbins & Steven Robbins. *Practical Unix Programming: A Guide to Concurrency, Comunication, and Multithreading*. Prentice-Hall, 1996.

[Tanenbaun_1] Andrew Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995.