INTRODUÇÃO AOS SISTEMAS OPERACIONAIS outubro/2013

SEMANA 13

Conceitos sobre threads.
Visão geral, threads de usuário e de kernel, modelos de multithreading, ciclo de vida. Exemplos nos sistemas operacionais.

1 - Introdução

Thread [linha de execução] é um fluxo independente de execução pertencente a um mesmo processo. A ideia básica sobre threads é que elas são processos leves [ligthweight processes].

Um processo é um programa em execução que contém um fluxo único de execução. Uma thread é também um programa em execução, porém com múltiplos fluxos de execução.

Tradicionalmente, cada processo tem seu espaço de endereçamento individual e apenas um fluxo de execução. Mas com o uso de threads, pode haver mais de um fluxo de execução no mesmo espaço de endereçamento, e com isso obter execução paralela.

2 - Conceitos sobre threads

Um processo tem duas partes: a ativa que é o fluxo de controle, e a passiva que é o espaço de endereçamento.

As threads são conhecidas como processos leves, pois possuem somente o fluxo de controle, e por isso consomem menos recursos do sistema.

O tempo necessário para criação e escalonamento de threads é menor quando comparado ao caso dos processos. O compartilhamento de memória entre as threads maximiza o uso dos espaços de endereçamento e torna mais eficiente o uso destes dispositivos. Como num processo todas as threads tem exatamente o mesmo espaço de endereçamento, elas também compartilham as mesmas variáveis globais, e uma thread pode acessar qualquer posição de memória dentro do espaço de endereçamento do processo. E já que normalmente as threads são criadas para cooperar e não competir, uma thread pode ler, escrever ou até apagar informações usadas por outra thread, sem um meio de proteção para isso.

O escalonamento entre threads acontece da mesma forma que entre processos. Portanto, cada thread tem o mesmo contexto de software e compartilha o mesmo espaço de memória de um

mesmo processo pai. Contudo, o contexto de hardware de cada fluxo de execução é diferente. Consequentemente, o tempo “perdido” (overhead) com o escalonamento das threads é muito menor do que o escalonamento de processos.

Muitas vezes os processos usam dados compartilhados, e o uso de várias threads [multithread] no mesmo espaço de endereçamento é mais eficiente e econômico que múltiplos processos.

Cada processo conta com uma estrutura de controle razoavelmente sofisticada. Nos casos onde se deseja realizar duas ou mais tarefas simultaneamente, o solução trivial à disposição do programador é dividir as tarefas a serem realizadas em dois ou mais processos, que implica na criação e manutenção de duas estruturas de controle distintas para tais processos. Isso onera o sistema e complica também no compartilhamento dos recursos, pois se trata de processos distintos.

Uma alternativa ao uso de processos comuns é o emprego de threads. Enquanto cada processo tem um fluxo único de execução, ou seja, só recebe a atenção do processador de forma individual, se um processo for divido em threads cada uma das threads componentes receberá a atenção do processador como um processo comum. E como só existe uma estrutura de controle de processo, o espaço de memória é o mesmo e todos os recursos associados ao processo podem ser compartilhados de maneira bem mais simples entre as threads componentes.

Segundo Tanenbaum, "as threads foram inventadas para permitir a combinação de paralelismo com execução sequencial e chamadas de sistema bloqueantes".

Desta forma, as threads podem ser entendidas como fluxos independentes de execução pertencentes a um mesmo processo, que requerem menos recursos de controle por parte do sistema operacional. Por isso é que as threads são consideradas processos leves [lightweight processes] e constituem uma unidade básica de utilização do processador.

Uma thread é criada e existe dentro do contexto de um processo. Isso significa que a thread e o seu processo pai compartilham a mesma memória, as mesmas variáveis, o mesmo descritor de arquivo, etc., e não requer o uso de código adicional para que haja comunicação entre todas as threads de um mesmo processo.

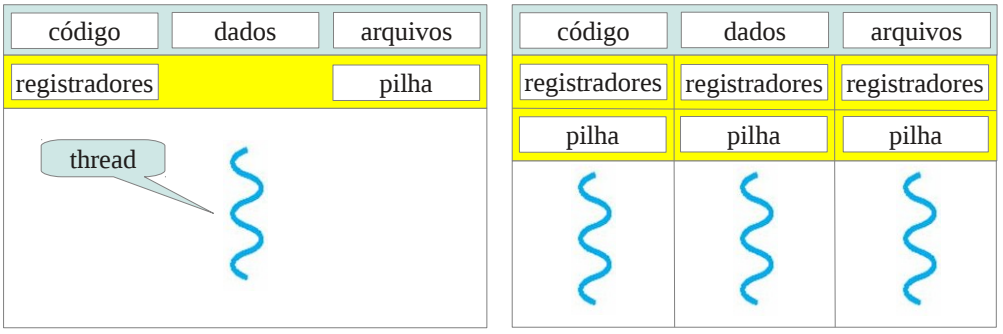

Fig 1 – processo tradicional com uma thread (a) e multithread (b)

Uma das vantagens no uso de threads está no fato do processo poder ser dividido em mais de um fluxo independente de execução. Por exemplo, enquanto uma thread está esperando

determinado dispositivo de I/O ou qualquer outro recurso do sistema, o processo como um todo não precisa ser bloqueado pois se uma thread entra no estado de bloqueio outra thread do mesmo processo pode estar aguardando na fila de processos prontos para continuar a execução do processo.

3 - Visão geral sobre threads

Normalmente, os processos são criados com uma única thread e a partir desta são criadas novas threads através de chamadas tipo *thread_create*( ). Nessa criação, geralmente é passado como parâmetro um nome de um procedimento para informar o que a thread vai executar, e como resultado sai um identificador ou nome para a thread criada. Ao finalizar o seu trabalho a thread chama uma função do tipo *thread_exit*( ) e desaparece para não ser mais escalonada.

Alguns benefícios no uso das threads são:

i. velocidade de criação: comparadas aos processos, as threads são mais fáceis de criar e destruir pois não tem quaisquer recursos associados a elas. Criar uma thread é muito mais rápido que criar um processo, e a troca de contexto é mais rápida quando comparada ao processo;

ii. capacidade de resposta: a utilização do multithreading permite que um programa continue executando e respondendo ao usuário mesmo se parte dele está bloqueada ou executando uma tarefa demorada. Por exemplo, enquanto um navegador web [por exemplo, Firefox] carrega uma página numa aba, noutra aba imprime e interage com o usuário;

iii. compartilhamento de recursos: todos os recursos alocados e utilizados pelo processo aos quais pertencem são compartilhados pelas threads;

iv. economia: como as threads compartilham recursos dos processos aos quais pertencem, é mais econômica a criação e realização de troca de contexto em threads;

v. utilização de multiprocessamento: é possível executar cada uma das threads criadas para um mesmo processo em paralelo, usando processadores (ou núcleos) diferentes, que aumenta bastante os benefícios do esquema multithreading;

vi. desempenho: existe ganho em desempenho quando há uso intensivo dos recursos da máquina (ex: CPU, I/O, etc.), pois as threads permitem que atividades se sobreponham e melhorem o desempenho da aplicação.

Threads que são vinculadas à CPU [CPU bound] têm sua velocidade de execução ligadas diretamente à velocidade da CPU. Por exemplo, uma thread pode ser considerada CPU bound quando executa um cálculo numérico sem realizar nenhuma operação de I/O tal como escrever na tela, no disco ou mesmo esperar pela entrada de teclado.

Threads que são vinculadas à I/O [I/O bound] estão no oposto do espectro quando comparadas a threads vinculadas à CPU. Isso significa que a velocidade de execução das threads I/O bound não estão ligadas à velocidade da CPU, e por isso usam poucos ciclos de CPU durante a

sua execução. Como exemplo, numa aplicação que espera que o usuário dê alguma entrada via teclado para continuar com o processamento, quase todo o tempo da thread é gasto esperando pela interação humana, e enquanto isso a CPU está ociosa ou atendendo a outras threads ou processos.

Em arquiteturas monoprocessadas o paralelismo obtido na execução de threads vinculadas à CPU é aparente, pois a CPU fica alternando entre cada thread de forma tão rápida que cria a ilusão de paralelismo real. Portanto, neste caso, a execução das threads acontece de forma concorrente assim como os processos.

Porém nem tudo é vantagem. Uma desvantagem é a insegurança devido ao compartilhamento do espaço de endereçamento. Outro problema está na baixa robustez dessa construção. Por exemplo, em múltiplos processos, se um deles tiver de ser finalizado os demais não serão afetados, já num multithread a paralisação de uma thread poderá afetar o processo, que irá comprometer todas as demais threads.

4 - Threads de usuários e de kernel

Sistemas computacionais que oferecem suporte para as threads são usualmente conhecidos como sistemas multithreading. Os sistemas multithreading podem suportar as threads segundo dois modelos diferentes, *user threads* e *kernel threads*:

i. user threads: as threads de usuário são aquelas oferecidas através de bibliotecas específicas e adicionais ao sistema operacional, que são implementadas acima do kernel utilizando um modelo de controle que pode ser distinto do sistema operacional, pois não são nativas neste sistema;

ii. kernel threads: as threads de sistema são aquelas suportadas diretamente pelo sistema operacional e, portanto, nativas.

Em sistemas não dotados de suporte a threads nativamente, o uso de bibliotecas de extensão permite a utilização de pseudo-threads. Exemplos de bibliotecas de suporte a threads de usuário são pthreads [POSIX threads], Win32 threads e Java threads.

Exemplos de sistemas operacionais que suportam kernel threads (threads de sistema) são Windows XP/2000, Solaris, Linux e Mac OS X.

Através das biblioteca de extensão são oferecidos todos os recursos necessários para a criação e controle das threads. Usualmente os mecanimos de criação de threads de usuário são bastante rápidos e simples, mas existem desvantagens. Por exemplo, quando uma thread é bloqueada pela espera de recursos de I/O, as demais threads frequentemente também o são devido ao suporte não nativo. Quando o suporte é nativo, a criação das threads é usualmente mais demorada, mas não ocorrem os inconveniente decorrentes do bloqueio de uma ou mais threads em relação às demais.

5 - Modelos de multithreading

Modelos de multithreading estão relacionados à forma como as threads são disponibilizadas para os usuários. São comuns três modelos distintos:

i. modelo *n* para um [many to one]: este modelo é empregado geralmente pelas bibliotecas de suporte de threads de usuário, onde as várias threads do usuário (*n*) são associadas a um único processo suportado diretamente pelo sistema operacional;

ii. modelo um para um [one to one]: modelo simplificado de multithreading verdadeiro, onde cada thread do usuário é associada a uma thread nativa do sistema. Este modelo é empregado em sistemas operacionais tais como Windows NT/2000 e IBM OS/2;

iii. modelo *n* para *m* [many to many]: modelo mais sofisticado de multithreading verdadeiro, onde um conjunto *n* de threads de usuário é associado a um conjunto *m* de threads nativas do sistema, não necessariamente do mesmo tamanho [*n* diferente de *m*]. Este modelo é empregado no sistema operacional Solaris e também suportado no GNU Portable threads.

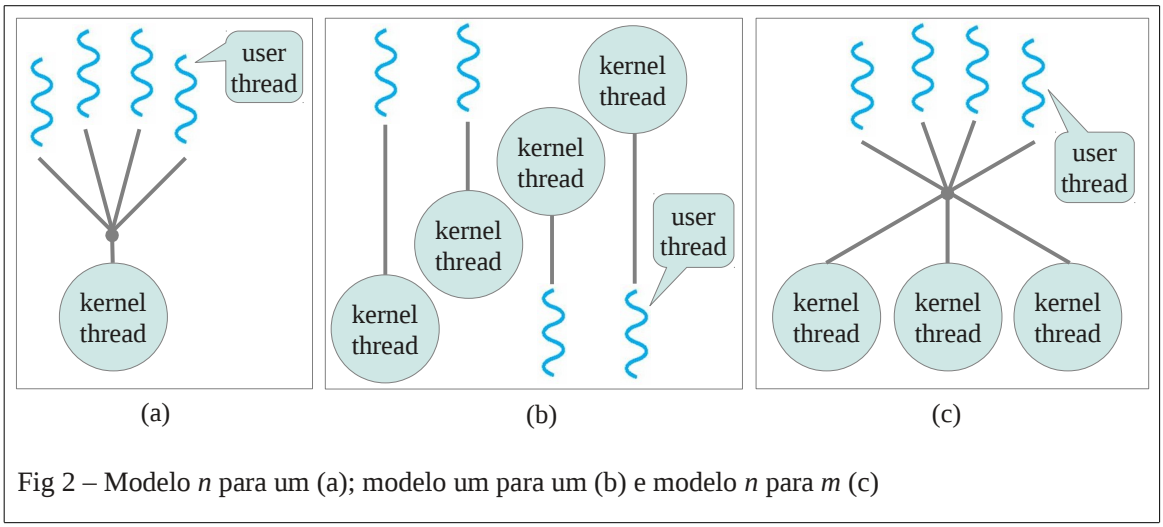

Fig 2 – Modelo *n* para um (a); modelo um para um (b) e modelo *n* para *m* (c)

6 - Ciclo de vida

Assim como os processos, as threads possuem estados durante o ciclo de vida, que são os mesmos discutidos em processos. Uma BCT (Bloco de Controle da Thread, Thread Control Block) ou tabela de threads deve então ser mantida para armazenar informações individuais de cada fluxo de execução. A BCT, semelhante ao BCP (Bloco de Controle do Processo, Process Control Block ou Process Descriptor), contém informações sobre:

i. o endereço da pilha;
ii. o contador de programa[1];

1 Contador de programa: (PC, Program Counter) é um ponteiro para a área de programa e tem por função armazenar o endereço (posição) da próxima instrução a ser executada na sequência de execução pelo processador.

iii. o registrador de instruções[2];
iv. os registradores de dados, endereços, flags;
v. os endereços das threads filhas;
vi. o estado de execução.

E para o processo, como um todo, restam informações do tipo endereço da área de trabalho, variáveis globais, apontadores para informações de arquivos abertos, endereços de processo filhos, sinais, semáforos[3] e de contabilização.

Threads podem comunicar-se através das variáveis globais do processo que as criou. A utilização destas variáveis pode ser controlada através de primitivas de sincronização [semáforos, monitores[4], ou construções similares]. Primitivas existem para bloqueio do processo que tenta obter acesso a uma área da memória que está correntemente sendo utilizada por outro processo. Primitivas de sinalização de fim de utilização de recurso compartilhado também existem. Estas primitivas podem "acordar" um ou mais processos que estavam bloqueados.

Threads permitem um paralelismo de granularidade mais fina, que é o paralelismo de instruções. No paralelismo de aplicações, que é o equivalente para o caso dos processos, a unidade de cálculo realizado de maneira concorrente á maior.

A granularidade de uma thread pode ser, por exemplo, um método ou um conjunto de instruções dentro de um programa. Para isso ser possível, cada thread mantém um contador de programa [PC, Program Counter] e um registrador de instruções [IR, Instruction Register] para dizer qual instrução é a próxima a ser executada e qual está sendo processada, respectivamente.

Além de registradores que contém suas variáveis atuais de trabalho, pilha com o histórico das instruções executadas e o estado de cada variável, para que uma thread possa ser executada ela deve pertencer a algum processo, ou seja, um processo deve ser criado anteriormente. Dessa forma, podemos dizer que processos são usados para agrupar recursos [espaço de endereçamento com o código do programa, variáveis globais, etc.] enquanto as threads são as entidades escalonadas pela CPU para a execução das tarefas.

2 Registrador de Instruções: (RI) é quem armazena a instrução que está sendo executada. A Unidade de Controle usa esta instrução para gerar sinais de controle para executar as operações determinadas na instrução.

3 Semáforos: quando dois processos concorrem pelo uso do mesmo recurso, é necessário usar sinalização para garantir que apenas um acesse o recurso de cada vez. Exclusão mútua é a expressão usada para dizer qua apenas um poderá acessar o recurso de cada vez. Uma forma de implementar exclusão mútua é através de semáforos, que são variáveis especiais que admitem apenas dois estados: UP e DOWN. Quando são usados semáforos, as chamadas de sistema realizam um controle extra para verificar o valor do semáforo.

4 Monitores: o monitor tem o mesmo propósito do semáforo, porém com uma construção mais ampla, pois é composto de funções e estrutura de dados próprios. Por exemplo, se fosse necessário usar mais de um semáforo para controlar tanto exclusão mútua quanto acesso a recurso compartilhado, dependendo da ordem poderia acabar bloqueado.