Instituto de Ciências Matemáticas e de Computação

SCE-217 Programação Concorrente
Profa. Dra. Regina Helena Carlucci Santana

Programação Paralela Utilizando Threads

Nome: Clerber Soares Mori
Nome: Leandro Botaro Martins
Nome: Luis Gustavo Pinheiro Machado
Nome: Ricardo dos Reis Cardoso

Índice

1. Introdução

Um thread é uma sequência de instruções que vão ser executadas num programa. No ambiente UNIX, os threads encontram-se dentro de um processo, utilizando os recursos desse processo. Um processo pode ter vários threads.

Pode-se dizer que um thread é um procedimento que é executado dentro de um processo de uma forma independente. Para melhor perceber a estrutura de um thread é útil entender o relacionamento entre um processo e um thread. Um processo é criado pelo sistema operativo e podem conter informações relacionadas com os recursos do programa e o estado de execução do programa:

- Process ID, process group ID, group ID
- Ambiente
- Directoria de trabalho
- Instruções do programa
- Registos
- Pilha (Stack)
- Espaço de endereçamento comum, dados e memória
- Descritores de ficheiros
- Comunicação entre processos (pipes, semáforos, memória partilhada, filas de mensagens)

Os threads utilizam os recursos de um processo, sendo também capazes de ser escalonados pelo sistema operativo e serem executados como entidades independentes dentro de um processo.

Um thread pode conter um controlo de fluxo independente e ser escalonável, porque mantêm o seu próprio:

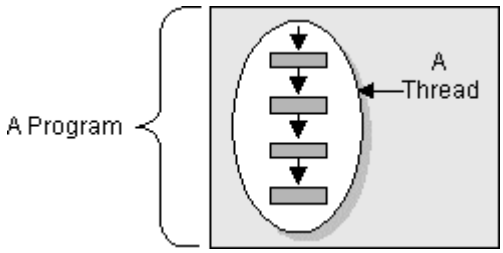
- Pilha
- Propriedades de escalonamento
- Dados específicos do thread

Um processo pode ter vários threads, partilhando cada um os recursos do processo e são executados no mesmo espaço de endereçamento. Com vários threads, temos em qualquer instante vários pontos de execução.

2. O que é um Thread?

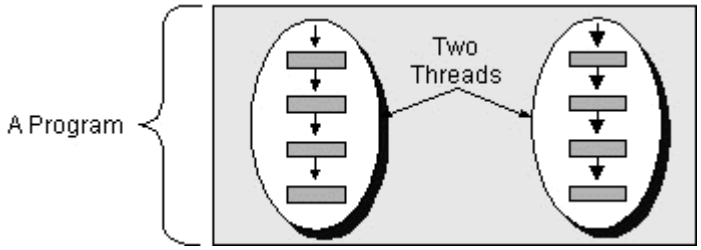
Todo programador esta familiarizado com programas escritos seqüencialmente. Provavelmente você já escreveu um programa que imprime "Hello World!" ou ordena uma lista de nomes ou calcula uma lista de números primos. Estes programas são seqüenciais. Ou seja, cada um tem um inicio, uma seqüência de execução, e um fim. A execução esta em um único ponto do programa, em qualquer instante de sua execução.

Um thread é similar aos programas seqüenciais descritos anteriormente. Um thread único também tem um inicio, uma seqüência, e um fim, e a qualquer instante durante a execução de um thread há um único ponto de execução. Entretanto, um thread por si só não é um programa; ele não pode rodar sozinho. Porém, ele roda "dentro" de um programa. A figura a seguir mostra esse relacionamento:



Definição: Um thread é um fluxo seqüencial de controle dentro de um programa.

Não há nada de novo no conceito de um único thread. O que realmente há de interessante acerca de threads não é um thread seqüencial único. É o uso de múltiplos threads em um programa, rodando ao mesmo tempo e tratando de diferentes tarefas. Isso é ilustrado na figura a seguir:



O Web browser HotJava Web é um exemplo de uma aplicação multithreaded. Dentro do browser HotJava você pode rolar uma página enquanto esta baixando um applet ou uma imagem, passa uma animação e toca uma música concorrentemente, imprime uma página no plano de fundo enquanto faz download de uma nova página ou assiste três algoritmos de ordenação competirem para terminarem mais rapidamente.Na sua vida você age de maneira concorrente... Por que seu browser não agiria?

Um thread é similar a um processo real à medida que um thread e um programa em execução são ambos fluxos seqüenciais únicos de controle. Entretanto, um thread é considerado lightweight porque ele roda no contexto do processo que o criou e toma vantagem de recursos alocados para o programa e de seu ambiente.

Como um fluxo seqüencial de controle, um thread deve possuir alguns recursos próprios dento de um programa em execução. (ele deve possuir sua própria pilha de

execução e seu próprio contador de programa, por exemplo.) O código rodando dentro de um thread apenas funciona dentro de seu contexto. Dessa forma, alguns textos utilizam *execution context* como sinônimo de thread.

Threads são freqüentemente chamados de lightweight processes e apesar deste termo ser bem simplificado, é um bom ponto inicial. Threads são primos dos processos do Unix, apesar de não serem realmente processos. Para entender a diferença, precisamos examinar a relação entre processos UNIX e tarefas e threads do Mach. No UNIX, um processo contém um programa em execução e um conjunto de recursos como tabelas de descrição de arquivos e espaço de endereçamento. No Mach, uma tarefa contém somente um conjunto de recursos, os threads tratam de todos os detalhes da execução. Uma tarefa Mach pode conter um número qualquer de threads associados a ela, e todos os threads devem estar associados com alguma tarefa. Todos os threads associados com uma determinada tarefa compartilham os recursos dela. Portanto um thread é essencialmente um contador de programa, uma pilha, e um conjunto de registradores – todas as outras tarefas pertencem à tarefa. Um processo Unix no Mach é modelado como uma tarefa com um único thread.

Como os Threads são bem pequenos se comparados com processos, a sua criação é relativamente barata em termos de custo de processamento. Processos precisam de seu próprio conjunto de recursos, já os Threads os compartilham. Os Threads Mach dão ao programador a habilidade de escrever aplicações concorrentes que rodam em máquinas com um ou mais processadores, transparentemente, usando processadores adicionais quando estes estiverem disponíveis. Além do mais, Theads podem aumentar a performance em um ambiente com um processador quando a aplicação executa operações que podem causar atrasos ou bloqueios, como arquivos e E/S de sockets.

3. Tipos de Threads

Existem duas famílias principais de threads:

POSIX-style threads, que geralmente rodam nos sistemas Unix.
Microsoft-style threads, que geralmente rodam nos PCs.

Futuramente, estas famílias serão subdivididas.

3.1 POSIX-style Threads

Essa família possui três subgrupos:

- "Real" POSIX Threads, baseado no IEEE POSIX 1003.1c-1995 padrão (também conhecido como ISO/IEC 9945-1:1996), parte do ANSI/IEEE 1003.1, edição de 1996. As implementações POSIX emergiram nos sistemas UNIX padrão.
  - Threads POSIX geralmente são referenciados como Pthreads.
  - Pode-se também encontrar Threads POSIX referenciados como POSIX.1c Threads.

    - Existem algumas referências que o chamam de “rascunho 10 do POSIX.1c”, que virou o padrão.
- Threads DCE são baseados no “rascunho 4” (um rascunho antigo) do padrão de threads do POSIX (que originalmente chamava-se 1003.4a, e virou 1003.1c depois da padronização). Pode-se acha-los em algumas implementações do UNIX.
- Unix International (UI) threads, também conhecidos como Solaris threads, são baseados no Unix International threads padrão (parente do POSIX standard). O único dos grandes variants de Unix que suportam UI threads são o Solaris 2, da Sun, e o UnixWare 2, da SCO.

Tanto os threads DCE como os threads UI são altamente compatíveis com os threads padrões do POSIX, apesar de ser necessário um certo trabalho para a conversão de algum desses tipos para o padrão “real” POSIX threads.

Os poucos distribuidores de pacotes Unix que ainda não implementam o padrão de threads POSIX pretendem implementá-lo logo que puderem.

Se estivermos desenvolvendo aplicações multithreads em Unix a partir do zero, seria melhor usar os threads do POSIX.

3.2 Threads Microsoft

Esta família de threads consiste de dois subgrupos, ambos desenvolvidos pela Microsoft.

Threads Win32 é o padrão utilizado no Windows 95 e Windows NT.

Threads OS/2 é o padrão utilizado no OS/2 da IBM.

Apesar de ambos terem sido implementados pela Microsoft, eles tomaram rumos diferentes ao passar dos anos. Passar de um para o outro requer uma certa quantidade de trabalho.

3.3 Outros

Mach e seus derivados (como o Digital UNIX) provêm um pacote de threads chamados “C Threads”. No entanto, não é largamente utilizado.

4. Programação Paralela Utilizando Threads

Os threads são muito usados para aplicações paralelas, já que eles são simples e leves de fazer dois ou mais processos rodarem paralelamente, ou pseudoparalelamente.

Os threads possuem a característica vista anteriormente de serem muito leves para criar, e compartilham de alguns recursos do sistema, como variáveis globais e outros. Assim, podemos ver que eles são uma opção superior em desempenho comparando-se com processos (fork).

Os threads são bem otimizados para serem usados em plataformas multi-processadas (SMP), tanto no Windows, Mac ou Unix, já que os sistemas operacionais de computadores com dois ou mais processadores fazem uma otimização destes recursos. Em computadores com mais de um processador, podemos ver que temos threads diferentes rodando em processadores diferentes, tendo assim uma execução realmente paralela.

Devido a sua velocidade de abertura, os threads são muito utilizados nos programas para esperar um dado de entrada, enquanto o thread principal executa outras operações.

Os threads são também muito bons para fazer programas paralelos uma vez que o seu funcionamento é similar a processos (fork), mas é mais leve. A sua vantagem em relação a extensões com o MPI ou PVM é que temos uma implementação mais simples (similar a fork) e nativa do sistema operacional.

A suas grande desvantagem é o fato de terem de rodar em uma mesma maquina. Porém a sua eficiência é grande.

Os programas paralelos, então, usufruem desta ferramenta para paralelizar vários pedaços do código sem gastar muitos recursos, e aproveitando um eventual paralelismo existente no hardware de maneira simples e nativa.

5. Sincronização de threads.

A seguir, vamos ver os métodos de sincronização de threads no padrão POSIX.

O padrão POSIX proce duas primitivas para sincronização de threads, o mutex e as variáveis de condição (condition variable).

Os mutexes são primitivas simples de travamento que podem ser usadas para controle de acesso a recursos compartilhados. Repare que com threads, todo o espaço de endereço é compartilhado, então tudo pode ser considerado como recurso compartilhado.

Porém, na maioria dos casos os threads funcionam individualmente com suas variáveis locais privadas, criadas de dentro de uma função chamada pelo pthread_create, e suas funções subseqüentes, podendo também acessar variáveis global.

Portanto, o acesso a itens que tem acessos em comum precisam ser controlados.

Veja o exemplo a seguir, de uma aplicação onde um único leitor e escritor se comunicam usando um buffer e fazem o controle de acesso usando mutex:

```
void reader_function(void);
void writer_function(void);

char buffer;
int buffer_has_item = 0;
pthread_mutex_t mutex;
struct timespec delay;

main()
{
```

```
  pthread_t reader;

  delay.tv_sec = 2;
  delay.tv_nsec = 0;

  pthread_mutex_init(&mutex, pthread_mutexattr_default);
  pthread_create( &reader, pthread_attr_default, (void*)&reader_function,
              NULL);
  writer_function();
}

void writer_function(void)
{
  while(1)
  {
      pthread_mutex_lock( &mutex );
      if ( buffer_has_item == 0 )
      {
          buffer = make_new_item();
          buffer_has_item = 1;
      }
      pthread_mutex_unlock( &mutex );
      pthread_delay_np( &delay );
  }
}

void reader_function(void)
{
  while(1)
  {
      pthread_mutex_lock( &mutex );
      if ( buffer_has_item == 1)
      {
          consume_item( buffer );
          buffer_has_item = 0;
      }
      pthread_mutex_unlock( &mutex );
      pthread_delay_np( &delay );
  }
}
```

Neste caso simples, o buffer pode armazenar somente um item, então ele tem somente dois estados, ou ele contém dados ou não.

O escritor primeiramente trava o mutex, e caso ele esteja travado, se bloqueando até que ele seja destravado. Então ele checa para ver se o buffer esta cheio. Caso ele esteja vazio, ele cria um novo item e seta o flag buffer_has_item, para que o leitor saiba que o

buffer tem um item. Ele, então destrava o mutex e espera por dois segundos para que o leitor tenha a chance de consumi-lo. Sem este delay, o escritor irá soltar o mutex e na próxima iteração já irá tentar pegá-lo de novo, para criar outro item. Seria muito provável que o consumidor ainda não tenha tido a chance de consumir o item, então este delay é necessário.

O leitor realiza a mesma idéia. Ele obtém o lock do mutex, checa para ver se existe um item criado, e se tiver, o consome. Ele, então, solta o lock do mutex e espera por um certo tempo (delay) para dar a chance do escritor criar um novo item. Neste exemplo, o leitor e consumidor irão rodar para sempre, criando e consumindo itens.

Porém, se um mutex não fosse mais necessário, ele deve ser destruído, usando a função pthread_mutex_destroy(&mutex).

A utilização correta dos mutexes nos garante a eliminação de race conditions. Porém, o mutex é uma primitiva fraca, já que ele tem somente dois estamos, travado e não travado.

As variáveis de condição (condition variable) vem para sanar esta deficiência por permitirem que outros threads se bloqueiem e esperem sinais de outros threads. Quando um sinal é recebido, o thread bloqueado é acordado e tenta obter o lock do respectivo mutex.

```
void print_message_function( void *ptr );

main()
{
  pthread_t thread1, thread2;
  char *message1 = "Hello";
  char *message2 = "World";

  pthread_create( &thread1, pthread_attr_default,
            (void*)&print_message_function, (void*) message1);
  pthread_create(&thread2, pthread_attr_default,
            (void*)&print_message_function, (void*) message2);

  exit(0);
}

void print_message_function( void *ptr )
{
  char *message;
  message = (char *) ptr;
  printf("%s ", message);
}
```

6. Conclusão

Os programas paralelos, então, usufruem desta ferramenta para paralelizar vários trechos do código sem gastar muitos recursos, e aproveitando um eventual paralelismo existente no hardware de maneira simples e nativa

7. Bibliografia

Gossamer **Threads** Inc.
www.gossamer-threads.com/

Threads Home
www.taunton.com/threads/index.asp

WebThreads
www.welshofer.com/WebThreads/

Common **Threads** of GenSource.com
www.gensource.com/common/

Getting Started With POSIX **Threads**
dis.cs.umass.edu/~wagner/threads_html/tutorial.html

SunOS 4.1 AnswerBook, “LightWeight Processes”, capítulo 2

M. L. Powell et al, “SunOS Multi-*thread* Architecture”, USENIX Winter 1991, págs. 1-14.

Tanenbaum, Andrew S., Modern Operating Systems, Prentice Hall 1992.