Programação Concorrente e Distribuída
THREADS

Professora: Thaís Vasconcelos Batista

2000.2

1 Introdução

Uma *thread* de controle, ou simplesmente *thread*, é uma sequência independente de execução do código de um programa dentro de um processo Unix.

Um processo Unix tradicional tem uma única thread que tem controle da memória do processo e outros recursos. É possível se ter mais de uma thread de controle ativa em um processo simultaneamente. As threads compartilham toda a memória do processo. Cada thread ler ou escreve na memória do processo ou faz solicitações para os serviços do sistema. Quando uma thread escreve na memória, as outras threads podem ler. As threads também compartilham todos os recursos do processo como descritores de arquivos, então mais que uma thread pode escrever no mesmo arquivo. Threads dentro de um processo são, em geral, invisíveis fora do processo.

As threads dentro de um processo são escalonadas e executadas independentemente da mesma forma que diferentes processos do Unix. Por exemplo, enquando uma thread executa uma solicitação de entrada/saída, ela não evita que outras threads sejam executadas. Em multiprocessadores, diferentes threads podem executar em diferentes processadores. Em monoprocessadores, threads devem ter sua execução intercalada. Tipicamente, uma outra thread começa a executar quando a thread corrente bloqueia ou usa o interface de tempo alocado para ela (também chamado de *timeslice* ou *time quantum*).

Com processos tradicionais do Unix, programadores que desejam ter mais que uma função ativa em um dado instante, usa vários processos. Os processos podem se comunicar com outros usando facilidades de comunicação entre processos (IPC), como pipes ou sockets. Os dados devem ser enviados do processo origem para o destino. O fluxo de dados é implicitamente controlado pela facilidade IPC, e erros em diferentes processos são isolados. Além disso, este mecanismo implica em duplicação de informações nos diferentes processos.

Se, ao invés de se usar processos, usar threads para cada função independente, a comunicação entre eles torna-se mais fácil; eles podem passar ponteiros para os dados ao invés de copiá-los.

Programação com mais de uma thread em um processo abre a possibilidade de uma thread interferir na memória ou recursos que outra thread está usando.

Este é um problema que deve ser tratado ao se trabalhar com threads.

2 Benefícios de Threads

- Throughput [1]

  Um programa com apenas uma thread de controle deve esperar toda vez que faz uma solicitação ao sistema operacional. Em sistemas monoprocessados assim como em um multiprocessador, usar mais de uma thread de controle torna possível que um processo sobreponha processamento com solicitações de entrada/saída. A thread que faz a solicitação espera, mas outra thread pode continuar o processamento. Isto permite que um processo mantenha pendentes várias solicitações de entrada/saída. Threads permitem esta sobreposição de solicitações de entrada/saída apesar de cada solicitação ser codificada no estilo síncrono.

  Outro exemplo de chamada síncrona é um programa que faz várias chamadas remotas de procedimentos (RPCs) para diferentes hosts para obter um resultado. Em um programa com única thread, os RPCs devem ser feitos serialmente. O programa deve enviar uma solicitação para um host e esperar pela resposta antes de enviar a próxima solicitação. Em um programa multithreaded, threads podem ser criadas para fazer cada solicitação RPC.

- Múltiplos processadores

  Computadores com mais de um processador podem ser aproveitados na programação com threads. Múltiplas threads oferecem uma maneira eficiente para se explorar o paralelismo de hardware. Um bom exemplo é um processo que faz a multimplicação de matrizes. Uma thread pode ser criada para cada processador disponível. Cada uma das threads repetidamente computa um único elemento do matriz resultante.

- Interface com Usuário

  Aplicações de interface gráfica com usuário que executam com única thread tipicamente exibe um *cursor de espera* (relógio de tempo) e congela enquanto a operação é realizada. Se tal aplicação for multithreaded, a seleção de um elemento de interface associado com a operação longa poderia fazer com que a operação fosse realizada por uma thread independente enquanto que a thread responsável pela monitoração da interface continua a executar. Isto permite que a aplicação e a interface com usuário permaneça disponível para outras solicitações.

- Servidores

  Programas servidores em aplicações cliente-servidor podem receber múltiplas solicitações simultaneamente. Estas solicitações podem ser atendidas de

[1] throughput = capacidade de processamento

maneira seriada se o servidor tem uma única thread de controle. Em uma situações de muitas solicitações, este comportamento ocasiona grande esperas por parte dos clientes. Servidores multithreaded podem atender mais rápido a demanda pois cada solicitação individual pode ser escalonada independentemente conforme seu grau de importância. Um bom exemplo desta estratégia é um servidor de banco de dados onde usuários concorrentes podem acessá-lo.

- Evitar Deadlock

  Em aplicações cliente-servidor, servidores podem também ser clientes de outros servidores. Suponha um servidor de banco de dados chamando um serviço de nomes que armazena suas informações em um servidor de banco de dados. Um servidor de banco de dados com única thread irá entrar em deadlock já que a única thread vai estar ocupada com a solicitação inicial. Um servidor multithreaded pode criar uma thread para cada solicitação.

- Estrutura de um Programa

  Threads podem ser usadas para simplificar a estrutura de um programa.

3 Programando com Threads

Esta seção apresenta um exemplo simples usando threads e introduz a interface de programação com threads.

`Exemplo:` Suponha uma aplicação que mostre uma série de figuras que estão em arquivos compactados. Normalmente, tal aplicação irá descompactar o arquivo, exibir a figura e esperar que o usuário indique que deseja ver a próxima figura (isto pode ser feito através de alguma opção da interface). Após o usuário solicitar a exibição da próxima figura, o programa começará a descompactar o arquivo. O usuário deve esperar até que o processo de descompactação seja concluído para poder visualizar a figura.

Usando threads, enquanto o usuário está vendo uma figura, o programa está descompactando a próxima. O exemplo abaixo ilustra esta simples aplicação de Threads.

```
int main(int argc, char *argv[])
{
   pthread_t fig_thread;
   int fileno;
   pic_buf_t * buf;

   for (fileno = 1; fileno<argc; fileno++) {
     if (fileno == 1) {
       /* primeiro arquivo */
       buf = get_next_picture((void *)argv[fileno]);
```

```
    } else {
      /* espera que a thread de descompressao finalize */
      pthread_join(pic_thread, (void **)&buf);
    }
    if (fileno < argc-1) {
      /* dispara uma thread para tratar a proxima figura (se houver) */
      pthread_create(&pic_thread, NULL,
         get_next_picture, (void *)argv[fileno + 1]);
    }
    display_buf(buf); /* mostra a figura corrente */
    free(buf);  /* libera o buffer */
    if (getchar() == EOF) /* espera que o usuario termine de olhar a figura */
       break;
    }
    return(0);
}

void *get_next_picture(void *arg)
{
   int fd;
   pic_buf_t *buf;

   fd = open((char *)arg, O_RDONLY);
   if (fd < 0)
        return (NULL);
   buf = (pic_buf_t *) malloc(...)
   while (nbytes = read(fd, ...)) != 0) {
        codigo para descomprimir arquivo e colocar em buf
   }
   close(fd);
}
```

3.1 Interfaces de programação com Threads

As interfaces que um programador deve utilizar para programar com Threads são:

- `Criando Threads` A interface para criar Threads é:

```
#include <pthread.h>

int pthread_create(
   pthread_t  *threadp,
   const pthread_attr_t *attr,
   void *(*start_routine)(void *), void *arg);
```

Figura 1: Interface para Criar Threads

```
void
pthread_exit(void *status);
```

Figura 2: Destruindo Threads

`pthread_create()` cria uma nova thread de controle que chama a função `start_routine()`, com um único argumento `arg`. Se `pthread_create()` tem sucesso, o `thread-id` para a nova thread é armazenado no ponteiro `threadp`, e `pthread_create()` retorna zero. Se houve algum problema com a criação da thread, `pthread_create()` retorna um erro. O argumento `attr` é usado para modificar os atributos da nova thread. Se `attr` é `NULL`, a nova thread é criada usando os atributos padrão.

- `Destruindo Threads` Quando uma thread finaliza seu trabalho, ela deve ser destruída. A maneira usual de se destruir uma thread é simplesmente retornar para a rotina que invocou. Porém, pode-se também fazer a chamada ilustrada na figura 2

- `Aguardando finalização da Threads` Uma maneira de determinar que outra thread completou sua função é esperá-la terminar. A função:

  int pthread_join(pthread_t thread, void **statusp);

  espera a finalização de uma thread (cuja thread-id é `thread`). Quando `thread` finaliza (ou se já finalizou), seu *completion status* é armazenado no ponteiro `statusp` e `pthread_join()` retorna 0. O *completion_status* é uma maneira conveniente de retornar os resultados da execução de uma thread. Por exemplo, no código do primeiro exemplo, a thread executando `get_next_picture()` retorna, como seu status de saída, um ponteiro para o buffer que contém as figuras descompactadas.

- `Recuperando Threads-id` Uma thread pode recuperar o seu `thread-id` chamando a função:

  pthread_t pthread_self();

- `Comparando Thread-id` Deve-se tratar thread-ids como objetos opacos. Por exemplo, é perfeitamente razoável para um sistema implementar uma `thread-id` como um tipo de dado estruturado. Por esta razão, não é possível comparar `thread-ids` usando os operadores de comparação de C. Para determinar se `threads-ids` são equivalentes chama-se a função:

  int pthread_equal(pthread_t t1, pthread_t t2);

  `pthread_equal()` retorna um valor diferente de zero se as duas `thread-ids` são a mesma. Se são diferentes, retorna zero.

- `Atributos de Thread`

  Threads contêm um conjunto de atributos que podem ser especificados quando a thread é criada. Atributos de threads são especificados através da estrutura `pthread_attr_t`. A estrutura contém os valores de vários atributos.

4 Sincronização entre Threads

Threads que cooperam para realizar algumas atividades podem interferir uma na outra já que elas acessam a mesma memória. A figura 3 ilustra esta interferência.

```
struct contas {
   int conta_corrente;
   int poupanca;
};

void
poup_para_contacorr(struct contas *c, int valor)
{
    c->poupanca -= valor;
    c->conta_corrente += valor;
}

int
total_balanco(struct contas *c)
{
    int balanco;

    balanco = c->conta_corrente + c->poupanca;
    return(balanco);
}
```

Figura 3: Exemplo que requer sincronização

```
Thread1                                 Thread2

balanco = poupanca+conta_corrente;       poupanca -= valor;
return(balanco);                           conta_corrente += valor;
```

Figura 4: Threads Concorrentes

Se a thread1 chama `total_balanco` enquanto thread2 chama `poup_para_contacorr`, os eventos ilustrados na figura 4 podem ocorrer.

Claramente, neste caso, o valor retornado por `total_balanco` será menor que a quantidade total de dinheiro nas contas.

A origem do problema é que mais que uma thread está tentando manipular o mesmo dado ao mesmo tempo. Este problema é chamado *Condição de Corrida*. Uma solução para este problema é permitir que apenas uma thread execute o código de cada vez. Isto é chamado de *Exclusão Mútua* pois a presença de uma thread exclui todas as outras. A seção de código que pode ser executada por apenas uma thread é chamada de *Seção Crítica*.

Exclusão mútua pode ser conseguida usando um *mutex*. Um mutex é uma estrutura de dados do type `pthread_mutex_t` alocada na memória. Há duas operações no mutex: bloqueio e desbloqueio.

Por exemplo:

A função `pthread_mutex_lock()` bloqueia o mutex. Outras threads que tentam

```
struct contas {
   int conta_corrente;
   int poupanca;
};

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void
poup_para_contacorr(struct contas *c, int valor)
{
    pthread_mutex_lock(&lock);
    c->poupanca -= valor;
    c->conta_corrente += valor;
    pthread_mutex_unlock(&lock);
}

int
total_balanco(struct contas *c)
{
    int balanco;

    pthread_mutex_lock(&lock);
    balanco = c->conta_corrente + c->poupanca;
    pthread_mutex_unlock(&lock);
    return(balanco);
}
```

Figura 5: Usando Mutex para Proteger uma Região Crítica

bloquear o mutex serão forçadas a esperar até que o mutex seja desbloqueado. Quando o mutex é desbloqueado, chamando `pthread_mutex_unlock()`, uma das threads que estava esperando pelo bloqueio ser liberado, irá bloquear o mutex.

Referências

[1] Kleiman, S. and Shah, D. and Smaalders, B. Programming with Threads. Sun Microsystems. 1996.

[2] Andrews, G. Foundations of Multithreaded, Parallel and Distributed Programming. Addison Wesley Inc. 2000.