Programação concorrente usando threads POSIX e Java

MAC 431 / MAC 5742

Introdução à Computação Paralela e Distribuída

Daniel Cordeiro

DCC - IME - USP

13 de agosto de 2012

Por que escrever programas concorrentes?

Desempenho para explorar arquiteturas paralelas com memória compartilhada (SMP, hyperthreaded, multi-core, NUMA, etc.)

Modelagem para descrever o paralelismo natural de algumas aplicações (tarefas independentes, sobreposição de operações de E/S, etc.)

Processos

- Cada processo tem seu próprio espaço de endereçamento:
  - *segmento de texto*: contém o código executável
  - *segmento de dados*: contém as variáveis globais
  - *pilha*: contém dados temporários (variáveis locais, endereços de retorno, etc.)
- O contador de programa e os registradores da CPU fazem parte do contexto do programa
- O contexto do programa é o conjunto mínimo de dados usados por um processo e deve ser gravado quando um processo é interrompido e relido quando um processo é retomado

Threads

- Permitem múltiplas atividades independentes dentro de um único processo
- *Threads* de um mesmo processo compartilham:
  - Todo o espaço de endereçamento, exceto a pilha, os registradores e o contador de programa
  - Arquivos abertos
  - Outros recursos
- Também são chamados de *processos leves*, pois a criação das *threads* e a troca de contexto são mais rápidas
- Permitem um alto grau de cooperação entre as atividades!

Veremos duas formas de programação

- POSIX *threads* (pthreads)
- *Threads* em Java

Padrão IEEE POSIX

- Norma internacional IEEE POSIX[1] 1003.1 C
- Apenas a API é normalizada, não a ABI
  - é fácil trocar a biblioteca que implementa a norma em tempo de compilação, mas não em tempo de execução
- *Threads* POSIX podem implementar *threads* em nível de usuário, em nível de kernel ou misto

[1] Portable Operating System Interface

Bibliotecas

LinuxThread (1996): em nível de kernel; padrão do Linux por muitos anos; não é totalmente compatível com o padrão POSIX

GNU Pth (1999): em nível de usuário; portável; POSIX

NGPT (2002): misto; baseado no GNU Pth; POSIX; não é mais desenvolvido

NPTL (2002): em nível de kernel; POSIX; é atualmente a biblioteca padrão do Linux

PM2/Marcel (2011): misto; compatível com POSIX; possui várias extensões interessantes para computação de alto desempenho (por exemplo, controle do escalonamento das *threads*)

Primeiros passos

```
$ gcc -v
Usando especificações internas.
Alvo: x86_64-linux-gnu
Configurado com: (...)
Modelo de thread: posix
versão do gcc 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)
```

Um programa em C deve conter

```
#include <pthread.h>
```

E para compilar é necessário incluir a opção -pthread

```
$ gcc -pthread programa.c
```

Exemplo – código-fonte

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid = (long)threadid;
    printf("Olá! Sou a thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("main: criando thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERRO; pthread_create() devolveu o erro %d\n", rc);
            exit(-1);
        }
    }

    for(t=0; t<NUM_THREADS; t++){
      pthread_join(threads[t], NULL);
    }
}
```

Exemplo – possível saída

```
$ ./exemplo1
main: criando thread 0
main: criando thread 1
main: criando thread 2
Olá! Sou a thread #0!
main: criando thread 3
Olá! Sou a thread #2!
main: criando thread 4
Olá! Sou a thread #3!
Olá! Sou a thread #1!
Olá! Sou a thread #4!
```

Considere o exemplo

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 5

int saldo = 1000;

void *AtualizaSaldo(void *threadid) {
  int meu_saldo = saldo;
  int novo_saldo = meu_saldo + (int)threadid*100;
  printf("Novo saldo = %d\n", novo_saldo);
  saldo = novo_saldo;
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        rc = pthread_create(&threads[t], NULL, AtualizaSaldo, (void *)t);
        if (rc) exit(-1);
    }

    for(t=0; t<NUM_THREADS; t++){
      pthread_join(threads[t], NULL);
    }

    printf("Saldo final = %d\n", saldo);
}
```

Ordem de execução

A ordem de execução não é garantida!

Thread 1	Thread 2	Saldo
Lê saldo: R$ 1.000		R$ 1.000
	Lê saldo: R$ 1.000	R$ 1.000
	Deposita R$ 300	R$ 1.000
Deposita R$ 200		R$ 1.000
Atualiza saldo R$ 1.000 + R$ 200		R$ 1.200
	Atualiza saldo R$ 1.000 + R$ 300	R$ 1.300

Solução: uso de sincronização (mutex)

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 5

pthread_mutex_t meu_mutex = PTHREAD_MUTEX_INITIALIZER;
int saldo = 1000;

void *AtualizaSaldo(void *threadid) {
   pthread_mutex_lock(&meu_mutex);
   saldo = saldo + (int)threadid*100;
   printf("Novo saldo = %d\n", saldo);
   pthread_mutex_unlock(&meu_mutex);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    long t;
    for(t=0; t<NUM_THREADS; t++)
        pthread_create(&threads[t], NULL, AtualizaSaldo, (void *)t);

    for(t=0; t<NUM_THREADS; t++)
      pthread_join(threads[t], NULL);

    printf("Saldo final = %d\n", saldo);
}
```

API básica de POSIX threads

Criação / destruição

- int **pthread_create**(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)
- void **pthread_exit**(void *value_ptr)
- int **pthread_join**(pthread_t thread, void **value_ptr)

Sincronização (semáforos)

- int **sem_init**(sem_t *sem, int pshared, unsigned int value)
- int **sem_wait**(sem_t *sem)
- int **sem_post**(sem_t *sem)
- int **sem_destroy**(sem_t *sem)

API básica de POSIX threads

Sincronização (mutex)

- int **pthread_mutex_init**(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
- int **pthread_mutex_lock**(pthread_mutex_t *mutex)
- int **pthread_mutex_unlock**(pthread_mutex_t *mutex)
- int **pthread_mutex_destroy**(pthread_mutex_t *mutex)

Sincronização (*conditions*)

- int **pthread_cond_init**(pthread_cond_t *cond, const pthread_condattr_t *attr)
- int **pthread_cond_wait**(pthread_cond_t *cond, pthread_mutex_t *mutex)
- int **pthread_cond_signal**(pthread_cond_t *cond)

Todo programa em Java executa várias threads

- Java foi a primeira linguagem de programação a incorporar threads desde a concepção da linguagem
- Se você já programou em Java, então já escreveu um programa multithreaded:
  - A máquina virtual mantém várias threads: thread main, threads do coletor de lixo, threads de finalização de objetos, etc.

Objetos do tipo Thread

Cada *thread* é associada a uma instância da classe `Thread`. Há duas estratégias básicas para usar esses objetos para criar aplicações concorrentes:

- controlar e gerenciar manualmente instâncias de `Thread`
- abstrair o gerenciamento de threads do resto da aplicação utilizando um *executor*

Definindo e criando threads

Uma aplicação que cria uma instância de `Thread` pode definir o código a ser executado concorrentemente de duas formas:

- usando herança, criamos uma nova classe que estende `Thread`
- escrevendo uma classe que implementa a interface `Runnable`

Usando herança

Uma classe que estende `Thread` deve sobrescrever o método **public void run()**

```
class Tarefa1 extends Thread {
  public void run() {
    for(int i=0; i<1000; i++)
      System.out.println("Usando herança");
  }

  public static void main(String args[]) {
    (new Tarefa1()).start();
  }
}
```

Usando a interface Runnable

A interface Runnable nos obriga a implementar o método **public void run()**:

```
class Tarefa2 implements Runnable {
  public void run() {
    for(int i=0; i<1000; i++)
      System.out.println("Usando Runnable");
  }

  public static void main(String args[]) {
    (new Thread(new Tarefa2())).start();
  }
}
```

Sleep, interrupções e joins

Thread.sleep() suspende uma *thread* por um período de tempo determinado. Útil para permitir que outras *threads* possam utilizar a CPU

Thread.interrupt() avisa uma *thread* que ela deve interromper o que está fazendo (por exemplo, terminar a execução). Uma *thread* que permite ser interrompida deve verificar periodicamente a *flag* de interrupção usando o método `Thread.interrupted()`. O modo mais comum de responder a uma interrupção é lançando uma `InterruptedException`

Thread.join() faz a *thread* atual esperar que uma outra *thread* termine sua execução

O modelo de consistência de memória

Suponha que duas *threads* A e B compartilhem um contador.

```
int contador = 0;
```

A incrementa o contador:

```
contador++;
```

Logo em seguida, B imprime o contador:

```
System.out.println(contador);
```

Qual a saída na tela?

- Se A e B fossem a mesma thread, certamente 1
- Mas a saída pode ser 0 ou 1; as mudanças em uma thread podem não ser visíveis para a outra. Sincronização estabelece uma relação de ordem entre as ações! (*happens-before relationship*)

Sincronização

A linguagem Java provê duas formas de sincronização básicas:

- métodos sincronizados
- expressões sincronizadas

Sincronização

A linguagem Java provê duas formas de sincronização básicas:

- métodos sincronizados
- expressões sincronizadas

```
public class ContadorSincronizado {
  private int contador = 0;

  public synchronized void incrementa() {
    contador++;
  }

  public synchronized int valor() {
    return contador;
  }
}
```

Sincronização

A linguagem Java provê duas formas de sincronização básicas:

- métodos sincronizados
- expressões sincronizadas

```
public void incrementa() {
  synchronized(this) {
    contador++
  }
}
```

Sincronização

A linguagem Java provê duas formas de sincronização básicas:

- métodos sincronizados
- expressões sincronizadas

```
public void incrementa() {
  synchronized(this) {
    contador++
  }
}
```

As sincronizações são reentrantes

Uma *thread* não pode pegar o *lock* de uma outra *thread*, mas pode usar um mesmo *lock* já adquirido mais de uma vez.

wait(), notify() e notifyAll()

Estes métodos da classe `Object` implementam o conceito de monitores sem utilizar espera ativa. Ao invés disso, notificam as threads indicando se estas devem ser suspensas ou se devem voltar a ficar em execução.

O lock do objeto chamado pela thread para realizar as notificações será utilizado. Por isso, antes de chamar um dos três métodos, o lock deve ser obtido utilizando-se o comando `synchronized`.

wait(), notify() e notifyAll()

Object.wait() suspende a *thread* que chamou o método até que outra *thread* a acorde ou até que o tempo especificado como argumento tenha passado

Object.notify() acorda, se existir, alguma thread que esteja esperando um evento neste objeto

Object.notifyAll() acorda todas as threads que estejam esperando neste objeto

wait()

```
synchronized (obj) {
  while (<condição não for satisfeita>)
    obj.wait(timeout);
     ... // Realiza ações que
         // assumem a condição satisfeita
}
```

notify()

```
synchronized (obj) {
  condição = true
  obj.notify()
}
```

Referências

pthreads

- Arnaud Legrand e Vincent Danjean. *How to Efficiently Program High Performance Architectures?* http://mescal.imag.fr/membres/arnaud.legrand/teaching/2011/PC_03_progpar.pdf
- *POSIX Threads Programming* https://computing.llnl.gov/tutorials/pthreads/

Java

The Java Tutorials: Concurrency: http://docs.oracle.com/javase/tutorial/essential/concurrency/