Sistemas Operacionais

Profª. Roberta Lima Gomes - email: soufes@gmail.com

2º Trabalho de Programação *(10,0)*

Data Entrega: 04/07/2016 (até 13:00h) Composição dos Grupos: até 3 pessoas

Material a entregar

- Por email: enviar um email para **soufes@gmail.com** seguindo o seguinte formato:
  - Subject do email: "**Trabalho 2**"
  - Corpo do email: lista contendo os nomes completos dos componentes da dupla em ordem alfabética
  - Em anexo: um arquivo compactado com o seguinte nome "**nome_do_grupo.extensão**" (*ex: joao-maria.zip*). Esse arquivo deverá conter todos os arquivos (incluindo *makefile*) criados para os programas C e Java. Vocês também devem incluir um arquivo README com as instruções sobre como rodar o programa. **Valendo ponto: clareza, endentação e comentários no programa.**

Descrição do Trabalho

Neste trabalho serão desenvolvidos dois programas, um em C e outro em Java, que, quando executados, trabalharão de forma cooperativa.

O programa em C deverá criar 5 threads (*thread 1* a *thread 5*) que serão responsáveis por enviar mensagens no formato "x-MSG-yyy", através de um *named pipe*, para o processo Java. Após enviar uma mensagem, cada thread deve esperar 1s para poder enviar novamente uma próxima mensagem. Observem que "x" no inínico da mensagem representa o identificador da thread que vai enviar a mensagem (*i.e.*, 1..5), e "yyy" representa um identificador único de mensagem com 3 dígitos (*i.e.*, 000..999). Esse identificador, além de ser único, deve ter uma relação com a ordem de criação das mensagens. Por exemplo, não pode existir duas mensagens "1-MSG-230" e "3-MSG-230", assim como certamente "5-MSG-170" foi criada antes de "2-MSG-863". Você deve implementar esse programa C utilizando a API Pthreads. Qualquer mecanismos de sincronização deverá ser implementado com base nessa API.

O programa em Java também será composto por 5 threads que deverão, de forma concorrente, ler as mensagens do *named pipe*. Além disso, vocês deverão implementar uma classe que deverá conter 2 buffers: um buffer "PAR" e um buffer "ÍMPAR". Essa classe deve conter dois métodos que serão utilizados pelas threads java para inserir as mensagens (extraídas do *named pipe*) nos buffers em função do identificador de cada mensagem. Isto é, se a mensagem apresentar um identificador par, a thread deve utilizar o método que insere no buffer PAR, e se a mensagem tiver um identificador ímpar, a thread deve utilizar o método que insere no buffer ÍMPAR. Para sincronização, vocês devem utilizar o conceito de monitor que os objetos java possuem.

> Importante: Vocês devem garantir o máximo de paralelismo possível! Pensem... é possível que duas (ou mais) threads escrevam simultaneamente nos buffers?

INFORMAÇÕES AUXILIARES SOBRE THREADS EM JAVA

O suporte para threads na linguagem JAVA é realizado através da classe *java.lang.Thread* cujos métodos definem a API disponível para a gestão de threads. Entre as operações disponíveis incluem-se métodos para iniciar, executar, suspender e gerenciar uma thread. O código que define o que uma thread vai fazer é o que estiver no método *run*. A classe *Thread* é uma implementação genérica de uma thread com um método *run* vazio, cabendo ao programador definir esse código para uma thread em particular.

O ciclo de vida de uma thread começa com a criação do respectivo objeto, continua com a execução do método *run* (iniciada através da invocação do método *start*), e irá terminar quando terminar a execução do método *run*.

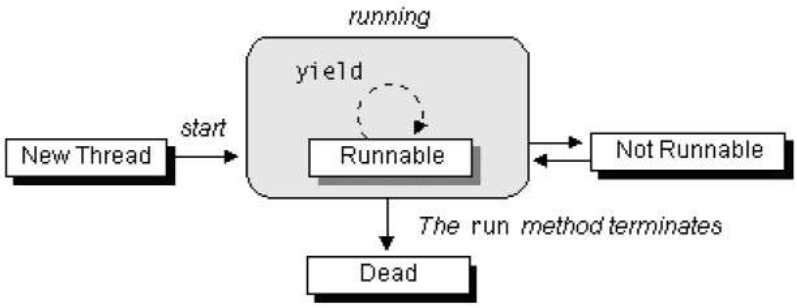

Figura 1: Ciclo de vida de uma thread

Umas das técnicas para criar uma nova thread de execução numa aplicação JAVA é criar uma subclasse de *Thread* e re-escrever o método *run*. A nova classe deverá redefinir o método *run* da classe *Thread* de forma a corresponder ao código que se pretende que a nova thread execute.

Exemplo 1

```
class PrimeThread extends Thread {

long minPrime;
PrimeThread(long minPrime) {
this.minPrime = minPrime;
}

public void run() {
// código a executar pela thread
. . .
}
}
```

A classe que pretendia criar a nova thread teria de executar o seguinte código:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

A outra forma de criar uma *Thread* é declarar uma classe que implementa a interface *Runnable* e que implementa o método *run.*

Exemplo 2

```
class PrimeRun implements Runnable {
long minPrime;
PrimeRun(long minPrime) {
this.minPrime = minPrime;
}

public void run() {
// código a executar pela thread
. . .
}
}
```

A classe que pretendia criar a nova thread teria de executar o seguinte código:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

A existência de threads concorrentes levanta a necessidade de sincronização. Em Java, cada objeto tem associado um mecanismo similar ao de um monitor através do qual é possível garantir o acesso exclusivo às seções críticas desse mesmo objeto. O controle de acesso às regiões críticas de um objeto é feito de forma automática pelo sistema de run-time de Java. O que o programador precisa de fazer é apenas assinalar as regiões críticas usando para tal a primitiva *synchronized*. Um bloco de código sincronizado de um objeto específico é uma região de código que só pode ser executada por uma thread de cada vez. Pode-se declarar um método como sendo *synchronized*:

```
synchronized int methodName () {... }
```

Ou pode-se declarar apenas uma parte de um método como sendo synchronized:

```
int methodName () {
  ...
  synchronized (this) {... }
  ...
}
```

Mais informações: vejam Tutorial de Java Threads em:
http://docs.oracle.com/javase/tutorial/essential/concurrency/

E para os mais “perdidos”, vejam um tutorial muito bom de Java:
http://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf