UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E APLICADAS
DEPARTAMENTO DE CIÊNCIAS EXATAS E APLICADAS

Nomes: Caio César Rangel Luciano e Livany Nunes Tavares
Matrículas 16.2.8180 e 16.2.8482
Disciplina: Sistemas Operacionais - CSI437
Professor: Samuel Souza Brito

Documentação Trabalho Prático

1 INTRODUÇÃO

Neste trabalho serão explorados o conceito e a utilização de threads, bem como problemas de sincronismo e mecanismos para resolvê-los. Para isso, foram utilizados os métodos de Peterson e Semáforo em ambiente Java.

2 CONDIÇÃO DE CORRIDA

a) Para esse exercício primeiramente criou-se uma classe chamada Counter a qual foi usada para incrementar o contador de 1 à 1000000. Por seguinte, implementou-se a classe RaceConditionalSimple cuja única função foi instanciar duas Threads (t1 e t2) que compartilhavam do mesmo contador da classe Counter e possuia o método resultado que apenas imprime o resultado do contador e é chamado na classe Testes.

O resultado obtido na maioria dos casos é incorreto! Isso acontece pois, supondo as duas Threads (t1 e t2), se t1 acessar o contador e incrementa-lo e antes que o resultado seja gravado t2 também acessa o contador e também o incrementa, o resultado gravado ao final será errôneo.

Ex: Count = 3

t1 faz count++ porém antes de gravar o resultado t2 também faz count++. O resultado será count == 4 e não igual a 5.

Classes usadas:

- Counter.java;
- RaceConditionalSimple.java;
- Testes.java.

b) O programa resolve o problema do Produtor-Consumidor por meio do algoritmo de Peterson. Cada Thread prepara para executar sua secção critica, mas indica o turno para a Thread concorrente. O ultimo turno sobreescrito será a Thread que irá acessar a seção crítica primeiro, e com isso incriemntar o contador. Enquanto uma das Threads acessa essa secção a outra aguarda em um while. Ao sair a Thread indica que terminou liberando a outra do while.

Optamos por colocar um do/while para contador menor que 100 para evitar loop infinito.

Classes usadas:

- RaceConditionalPeterson.java;
- Testes.java

3 PRODUTOR CONSUMIDOR

Para a problemática do produtor consumidor com 2 produtores e 3 consumidores optamos por resulver com uso de semáfaro.

Classes usadas:

- Consumer.java;
- Producer.java;
- Q.java;
- Testes.java.

A classe Consumer.java é responsável por determinar quantos itens os consumidores pegarão através de um for que itera até a quantidade de produtos que cada um vai pegar. Cada consumidor pega exatamente a mesma quantidade de itens.

De forma semelhante a classe Producer.java determina quantos produtos serão produzidos pelos produtores também através de um for. Cada produtor produz a mesma quantidade de produtos, um de cada vez.

Inicialmente são criados dois deles, semCon e semProd. O semCon é inicializado com zero para garantir que o semProd execute primeiro garantindo que haverá produto para o

consumidor. Além disso temos um método para consumir o item (get()) e um para produzi-lo (put(int item)).

O que ocorre é que o método get() é acessado e tenta fazer um semCon.acquire(), porém como semCont() é inicializado em zero ele fica em estado de espera. Concomitantemente o método pu(int item) é acessado faz um semProd.acquire() sinalizando que a secção crítica está ocupada (situação em que o item é produzido e colocado a disposição do consumidor). Ao sair o método ainda realiza um semCon.release() o que libera o estado de espera do consumidor no método get().

4 TRAVESSIA DA PONTE

O cenário desse problema girava em torno de organizar a travessia de carros em uma ponte de modo que não houvesse colisões. Desse modo, optamos por utilizar Monitor (método synchronized). No método entradaCarrosVermelhos() caso houvesse algum carro azul atravessando a ponte o método wait() era chamado de forma a fazer com que os carros vermelhos esperassem a ponte ficar livre. Situação inversa foi implementada no método entradaCarrosAzuis(), onde caso houvesse carros vermelhos passando pela fonte os azuis aguardavam pacientemente sua vez.

Para saber se havia carros atravessando a ponte e se ele é azul ou vermelho, criamos duas variáveis uma para armazenar o número de carros azuis que entram na ponte e outra que faz a mesma coisa para os vermelhos. Essas variáveis, instanciadas em zero, são incrementadas no método de entrada e decrementadas no de saída.

Quando todos os carros de uma das cores terminam de atravessar a ponte o método notifyAll() fica incumbido de destravá-la de modo a liberar o acesso para outros carros que desejem atravessar.

Tal solução evitava colisões porém esbarrava no caso de bloqueio indefinido (stravation) uma vez que enquanto houvessem carros de uma mesma cor atravessando a ponte os da cor restante esperavam ilimitadamente até todos passarem.

Desse modo, acrescentamos mais duas variáveis inteiras (esperaAzul e esperaVermelho) responsáveis por contar o numero de carros que esperam para passar na ponte. Além de uma variável booleana que é verdadeira quando está na vez dos carros azuis e falsa na fez dos vermelhos.

5 CONCLUSÃO

Com esse trabalho aprendeu-se que no ambiente de multiprogramação, vários processos podem acessar e manipular os mesmos dados simultaneamente e o resultado da execução depende da ordem específica em que o acesso ocorre, esse tipo de situação é chamado de condição de corrida. Para se proteger contra a condição de corrida, precisamos garantir que apenas um processo de cada vez possa manipular os dados. O cenário é mais claramente representado pelo problema da seção crítica. A solução da Peterson é uma solução para esse problema para 2 processos. Ferramentas de sincronização como o Semáforo podem lidar com mais processos, como acontece no problema do produtor-consumidor no com mais de um consumidor e um produtor.

Por fim, vale também ressaltar a facilidade em se deparar com stravation ao utilizar os métodos wait e notify, necessitando que algum critério seja tomado para que um dos envolvidos a esquemática não fique predicado com espera infinita.