UNIVERSIDADE ESTADUAL DO CEARÁ

CENTRO DE CIÊNCIAS E TECNOLOGIA – CCT

CIÊNCIA DA COMPUTAÇÃO – 2014.2

RELATÓRIO DO PROBLEMA DA CONTA POUPANÇA

DISCIPLINA: PROGRAMAÇÃO PARALELA E CONCORRENTE

[redacted]

MATRÍCULA: [redacted]

FORTALEZA, 2015

Introdução

O objetivo desse relatório é descrever a aplicação dos conhecimentos e técnicas aprendidos durante a disciplina de Programação Paralela e Concorrente – 2014.2, através do problema da Conta Poupança, definido previamente no início da cadeira. O relatório contém informações sobre o problema proposto, especificações sobre a implementação, política e técnicas de sincronização utilizadas, estrutura de classes adotadas, resultados obtidos e comentários sobre a experiência com o desenvolvimento.

Problema da Conta de Poupança

Imagine a seguinte situação: temos uma conta poupança compartilhada por várias pessoas. As pessoas (clientes) podem sacar ou depositar dinheiro continuamente. O saldo da conta é resultado das transações de saque e depósito (saldo = valor dos depósitos atuais – valor dos saques atuais).

A conta deve obedecer as seguintes especificações e restrições:

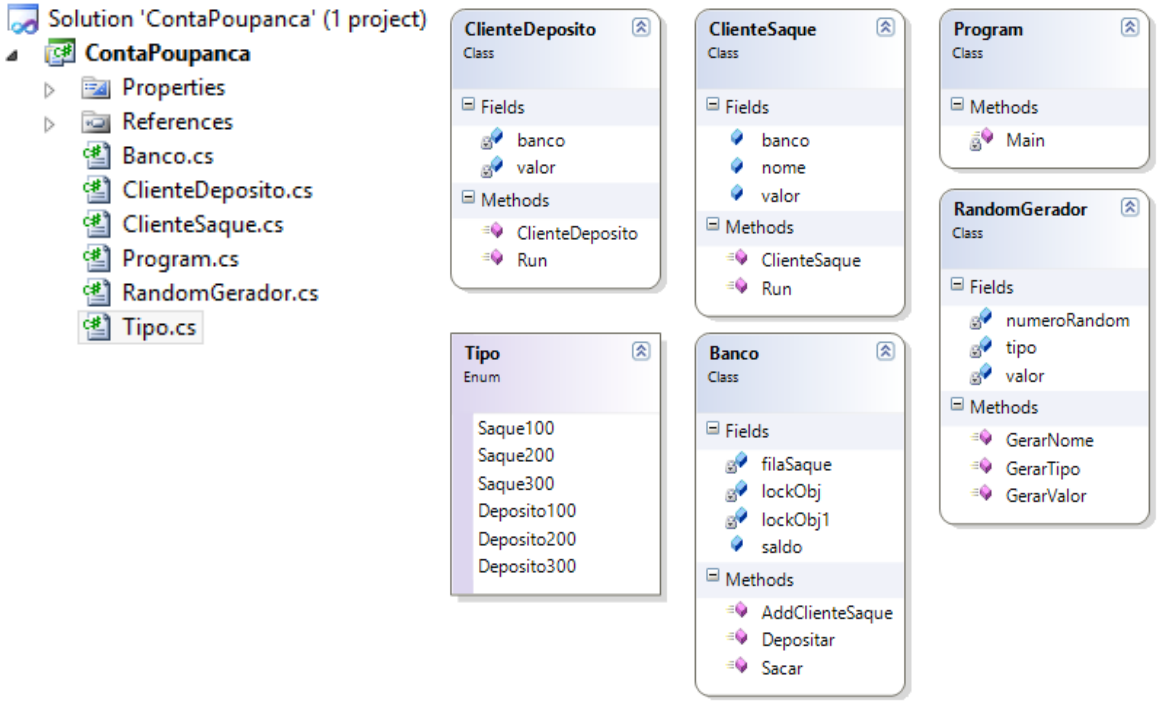
- O saldo não pode ser negativo (saldo >= 0, sempre);
- Os depósitos e os saques devem ser atendidos por ordem de chegada;
- Os clientes de saque devem seguir uma política de fila, onde o primeiro a chegar deve ser o primeiro a ser atendido – FIFO;
- Os clientes devem trabalhar com os valores de 100, 200 e 300 reais, tanto pra saque, como pra depósito;
- O intervalo entre as transações deve ser de 2 segundos (2000 ms);
- A fila de espera de saque não tem tamanho definido e não há fila para depósito;
- Os clientes de saque devem “dormir” quando estiverem esperando na fila sua vez.

Implementação

O projeto foi desenvolvido utilizando a linguagem de programação C#, com o ambiente de desenvolvimento Visual Studio 2010 sobre uma aplicação Console. Os mecanismos de sincronização foram os Monitores nativos da própria linguagem

(Monitor) e os métodos de controle das condições de sincronização Wait(lock) e PulseAll(lock) (análogos ao wait() e notifyAll() de Java, respectivamente) foram adotados.

O projeto ContaPoupanca foi criado, juntamente com as classes:



- Banco: contém os métodos base de sincronização dos clientes de saque (Sacar) e depósito (Depositar), juntamente com o métodos de adição de cliente sacador na fila definida na classe, através da seguinte linha de código:

```
private Queue<ClienteSaque> filaSaque = new Queue<ClienteSaque>();
```

- ClienteDeposito: com construtor implementado e método Run, que chama o método Depositar da instância de Banco (banco), presente em cada cliente.
- ClienteSaque: com construtor implementado e método Run, que chama o método AddClienteSaque, passando como parâmetro o cliente atual, e método Sacar presente da instância de Banco (banco) presente em cada cliente.
- Program (Principal): contém a Main e foi onde os mecanismos de criação das Threads foi implementado.
- Tipo: enum que contém cada tipo de cliente que possa vir a ser criado, tanto em relação a transação (Saque ou Depósito), como em relação aos valores (100, 200 e 300 reais). A saber:

```
public enum Tipo
{
 Saque100 = 0,
```

```
        Saque200 = 1,
        Saque300 = 2,
        Deposito100 = 3,
        Deposito200 = 4,
        Deposito300 = 5
    }
```

- RandomGerador: contém métodos de geração aleatória de nome das Threads, tipo das Threas, obedecendo ao enum Tipo e valores (100, 200 ou 300);

Mecanismo de Sincronização

A classe Banco contém os métodos sincronizados, ilustrados abaixo, que controlam o recurso de saldo da conta poupança, Sacar – referente ao controle dos ClienteSaque, e Depositar – referente ao controle dos ClienteDeposito.

```
class Banco{

        public void Sacar(int valor)
        {
            Monitor.Enter(lockObj);
            try
            {
               //Enquanto o valor de saque for maior que o valor de
//saldo disponível, ou saldo negativo, faça a thread requerente
//dormir; se condição foi atendida, acorde as threads
                while ( (valor > saldo || saldo <= 0))
                {
                    Monitor.Wait(lockObj);
                    Monitor.PulseAll(lockObj);
                }

                var sacadorAtual = filaSaque.First();

                //Se o valor a ser sacado estiver disponível
                if ( saldo > 0 && saldo >= valor )
                {
                    Console.WriteLine("Cliente " +
Thread.CurrentThread.Name + " solicitou Saque");
                    saldo -= valor;
                    Console.WriteLine("Saque R$ " + valor);
                    Thread.Sleep(2000);
                    Console.WriteLine("----------------------");
                    Console.WriteLine("Saldo Atual: " + saldo);
                    Console.WriteLine("----------------------");
                    filaSaque.Dequeue();
                    Console.WriteLine("Cliente " +
Thread.CurrentThread.Name + " efetuou o Saque.");
                    Monitor.PulseAll(lockObj);
                }
```

```
            }
            finally
            {
                Monitor.Exit(lockObj);
            }
        }

        public void Depositar(int valor)
        {
            Monitor.Enter(lockObj);
            try
            {
                Console.WriteLine("Cliente " +
Thread.CurrentThread.Name + " solicitou Deposito");
                Console.WriteLine("Deposito R$ " + valor);
                saldo += valor;
                Thread.Sleep(2000);
                Console.WriteLine("----------------------");
                Console.WriteLine("Saldo Atual: " + saldo);
                Console.WriteLine("----------------------");
                Monitor.PulseAll(lockObj);
            }
            finally
            {
                Monitor.Exit(lockObj);
            }
        }
    }
```

Nos códigos abaixo, presentes nos métodos, respectivamente Sacar e Depositar, temos que:

```
while ( (valor > saldo || saldo
<= 0))
{
      Monitor.Wait(lockObj);
      Monitor.PulseAll(lockObj);
}

/*Código Comentado*/

if ( saldo > 0 && saldo >= valor
)
{
      /*Código Comentado*/
      saldo -= valor;
      /*Código Comentado*/
      filaSaque.Dequeue();
      Monitor.PulseAll(lockObj);
}
```

```
saldo += valor;

/*Código Comentado*/

Monitor.PulseAll(lockObj);
```

O método Wait(lockObjt) faz com que, enquanto a condição para atender a thread de saque requerente não for válida, a thread durma, fazendo com que o acesso ao recurso não seja permitido indevidamente. Indevidamente por dois motivos: por que o valor para sacar é superior ao valor de saldo disponível na conta ou por que o saldo está no limite, e saldos negativos não efetivam transações de saque.

Depois que a condição for satisfeita, as Threads acordam e uma nova verificação é feita, agora no condicional if. Se for válido, o saque é permitido, o primeiro da fila, a thread que fez o pedido de saque, é removido por meio do método Dequeue() da fila e as threads que estão dormindo são acordadas, já que a transação de saque está liberando o recurso de saldo para novos clientes.

No método Depositar, como não há exigências sobre a política de chegada dos depósitos, o saldo é atualizado com o valor que cliente atual quer inserir na conta e, logo em seguida, libera o acesso pros demais clientes.

A política de criação das Threads foi especificada da seguinte forma:

- 20 Threads de cada Tipo (Saque de 100 reais, Saque de 200 reais, Saque de 300 reais, Depósito de 100 reais, Depósito de 200 reais e Depósito de 300 reais) devem ser criadas;
- A criação deve acontecer de forma aleatória.

O código abaixo da Main ilustra qual foi a estratégia adotada para a criação aleatória das threads. O controle, basicamente, é feito pela variável booleana

```
bool clientesCriados = false;
```

que indica, se verdadeira, que todos as threads necessárias foram criadas, e através do switch dos tipos criados randomicamente pelo método GerarTipo().

Se, por exemplo, o tipo Saque100 foi gerado e se ainda é possível criar thread do tipo Saque100 (valor máximo definido foi 20, que é decrementado a medida que os clientes daquele tipo são criados), então cria-se uma thread do tipo ClienteSaque, passando como valores pro construtor a instância do banco, o nome da thread – gerado randomicamente, e o valor para saque, que no caso seria 100.

O critério de parada para finalizar a criação das threads é a verificação se todas as variáveis inteiras que contabilizam os tipos criados estão zeradas. Se sim, a variável de controle `clientesCriados = true` e espera-se a finalização das threads, para o fim do programa.

```
namespace ContaPoupanca
{
    class Program
    {
        static void Main(string[] args)
        {
            //crie 20 clientes deposito de 100,200 e 300

            int intervaloProximoCliente = 2000;
            Banco banco = new Banco();
            RandomGerador randomGerador = new RandomGerador();

            int nSaque100 = 20;
            int nSaque200 = 20;
            int nSaque300 = 20;
            int nDeposito100 = 20;
            int nDeposito200 = 20;
            int nDeposito300 = 20;
            bool clientesCriados = false;

            while (!clientesCriados)
            {
                var tipo = randomGerador.GerarTipo();
                switch (tipo)
                {
                    case Tipo.Saque100:
                        if (nSaque100 > 0)
                        {
                            var nome1 = randomGerador.GerarNome(tipo.ToString(),
nSaque100, 100);
                            ClienteSaque clienteSaque1 = new ClienteSaque(banco,
nome1, 100);
                            Thread thread1 = new Thread(new
ThreadStart(clienteSaque1.Run));
                            thread1.Name = nome1;
                            thread1.Start();
                            Thread.Sleep(intervaloProximoCliente);
                            nSaque100--;
                        }
                        break;

/*Código Comentado*/

                    case Tipo.Deposito100:
                        if (nDeposito100 > 0)
                        {
                            var nome11 = randomGerador.GerarNome(tipo.ToString(),
nDeposito100, 100);
                            ClienteDeposito clienteDeposito1 = new
ClienteDeposito(banco, 100);
                            Thread thread11 = new Thread(new
ThreadStart(clienteDeposito1.Run));
                            thread11.Name = nome11;
                            thread11.Start();
                            Thread.Sleep(intervaloProximoCliente);
                            nDeposito100--;
                        }
                        break;
                }

                if (nSaque100 == 0 && nSaque200 == 0 && nSaque300 == 0 &&
                    nDeposito100 == 0 && nDeposito200 == 0 && nDeposito300 == 0)
```

```
                {
                    clientesCriados = true;
                }
            }
        }
    }
}
```

Execução

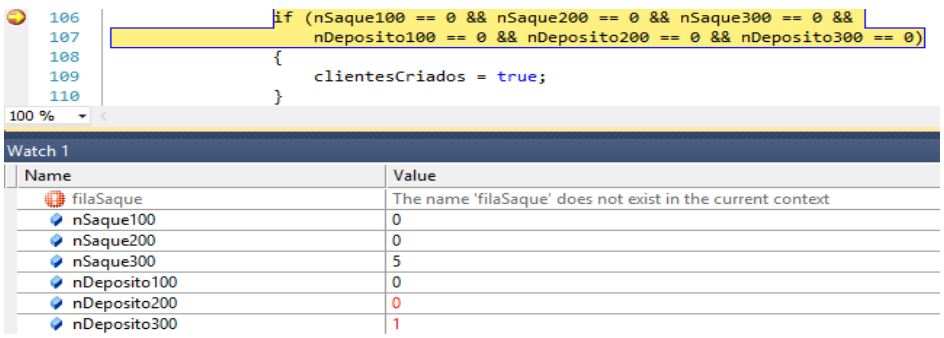
Início

Na figura acima, o programa começou com a criação de uma thread do tipo depósito de 100 reais, seguida de outra do tipo depósito de 300 reais, totalizando o saldo em conta de 400 reais.

Logo em seguida, a aleatoriedade de criação, gerou uma thread de saque de 200 reais, o cliente criado foi adicionado na fila e foi verificado se o saldo é valido para o saque requerido. Como o solicitado é de 200 reais e o saldo em conta é de 400, a thread Saque 200_20_200 realiza o saque, deixando o saldo em banco de 200 reais.

A thread de depósito 300_19_300 é criada e atualiza o saldo do banco, incrementando 300 reais, para 500 reais. Por fim, uma nova thread ,que irá solicitar saque de 300 reais, foi criada.

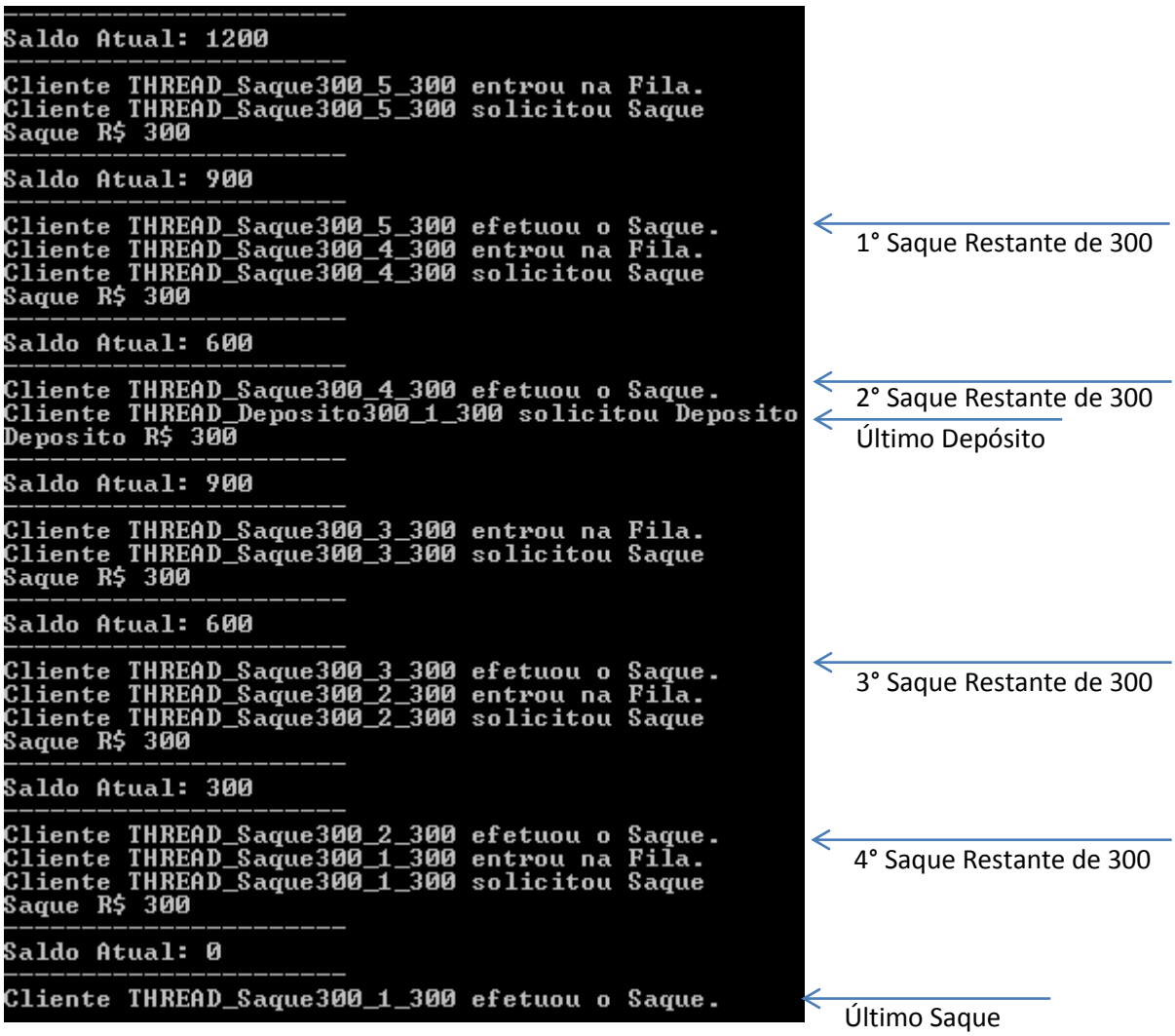
Fim



Na figura acima, o Breakpoint foi estrategicamente posicionado para indicar quantas threads ainda seriam criadas para poder acompanhar o tempo final de execução. Todos os clientes que solicitaram saque de 100 e 200 reais já foram criados e seus respectivos saques foram efetuados com sucesso e todos os clientes que requisitaram realizar depósitos de 100 e 200 reais conseguiram depositar com sucesso também. Ainda faltam 5 clientes realizarem saques de 300 reais e apenas 1 clientes efetuar um depósito de 300 reais.

Matematicamente, para o total de saques ainda restantes ser válido e considerando que o depósito restante de 300 reais já tenha sido feito, o saldo em conta deverá ser de 1500 reais (5 x 300 + 1x300). Caso o depósito restante de 300 reais ainda não tiver sido feito, o saldo em conta deve ser de 1200 reais (5x 300 - 1 x 300).

Como a figura abaixo ilustra, o saldo é de 1200 reais, logo o depósito de 300 ainda será feito antes do fim do programa, como identificado pela seta abaixo.



Experiência

Com o trabalho, pode-se colocar em prática os conhecimentos adquiridos na disciplina. O conceito de deadlock ficou claro em algumas situações de teste mais iniciais. Nas primeiras implementações, quando a transação de depósito não era realizada no começo da execução, deixando o saldo com o valor maior ou igual ao valor de saque que seria solicitado pelo primeiro cliente de saque que fosse criado, um deadlock em relação a todos os clientes saque que fossem criados surgia, fazendo com que os clientes entrassem na fila de espera e não saíssem mais.

O problema foi resolvido adotando-se a política de verificação constante que checa se o valor do saldo é viável para ser sacado, e maior que 0. Se não, os clientes dormem.

A ansiedade de ver as transações acontecerem e verificar quantas threads ainda faltavam para acabar com o processo todo gerou um entendimento maior da implementação. O ambiente que o Visual Studio oferece ajudou bastante para essas verificações.