Documentação do Projeto Bus-Thread

Processamento Paralelo

Bus-Thread

Simulação de um dia de aulas na Universidade Feevale

Eduardo Lorscheiter

Loreno Enrique Ribeiro

2025/02

1. Introdução

Thread, antes de qualquer coisa precisamos entender o que essa palavra representa, já que vamos usá-la em diversos momentos. Então, o que é uma thread?

Uma thread é a menor unidade de processamento que pode ser gerenciada por um sistema operacional. Ela permite que múltiplas tarefas sejam executadas concorrentemente, dentro de um mesmo processo, compartilhando os mesmos recursos e memória. No contexto da programação, basicamente uma thread permite com que nós separemos um processamento grande em pequenos processos, e esses pequenos processos agora podem ser processados ao mesmo tempo, isso faz com que o tempo de processamento de um programa consiga melhorar absurdamente.

Este trabalho tem como objetivo apresentar uma solução para o problema do ônibus, um clássico exemplo de sincronização em sistemas operacionais.

A solução foi desenvolvida utilizando threads, que permitem a execução simultânea de diferentes partes de um programa dentro de um mesmo processo. Com isso, é possível representar de forma realista o comportamento paralelo dos alunos e dos ônibus, além de aplicar técnicas de sincronização para evitar conflitos e garantir a ordem correta das ações.

Utilizamos threads para simular o comportamento simultâneo de alunos e ônibus dentro da Universidade Feevale. Os alunos podem estar em diferentes locais da Universidade, então eles levam tempos diferentes até chegar ao ponto de ônibus. Enquanto os alunos se deslocam até o ponto de ônibus, os ônibus chegam em momentos diferentes e os alunos que já chegaram ao ponto vão embarcando e indo embora, de forma paralela e sincronizada.

2. Requisitos do Projeto

O objetivo desse projeto é desenvolver um sistema que utilize threads para simular um dia de aulas da Universidade Feevale. Aqui vamos destacar alguns requisitos que recebemos na proposta do projeto e precisamos seguir e outros que foram definidos durante a execução da proposta.

Requisitos definidos na descrição da atividade (comportamentos essenciais):

- Alunos que chegam no ponto de ônibus devem aguardar o ônibus chegar;
- Quando o ônibus chega, apenas os alunos que já estavam aguardando no ponto podem embarcam;
- Alunos que chegarem no ponto durante o embarque do ônibus devem aguardar o próximo ônibus;
- Os alunos devem ser criados de forma aleatória e frequentar as aulas;
- Todas as salas devem iniciar e finalizar as aulas ao mesmo tempo.
- Duração da aula de cada sala deve ser aleatória (entre 2 e 10 minutos);
- Os ônibus devem ser gerados com uma frequência aleatória (entre 2 e 3 minutos);
- Os ônibus possuem uma capacidade máxima de 50 alunos;
- Se houver mais de 50 alunos na parada no momento que o ônibus chegar, deve embarcar 50 alunos, o excedente deve esperar o próximo ônibus;
- O ônibus vai embora imediatamente se não possuir alunos na parada;
- Utilizar threads para representar os alunos e os ônibus.

Requisitos definidos durante a execução (comportamentos de nossa escolha):

- A execução simulará apenas um dia de aulas;
- Tempo para o aluno chegar até a parada de ônibus (entre 30 segundos e 10 minutos);
- Tempo de embarque dos alunos no ônibus (1 segundo por aluno);
- O número de salas de aula é gerado aleatoriamente (entre 5 e 10);
- Os ônibus entram em circulação apenas quando acabam as aulas;
- Os ônibus continuarão em circulação até levar todos os alunos da Universidade Feevale embora.

3. Desenvolvimento

O desenvolvimento do projeto ocorreu a partir do entendimento detalhado do problema e da elaboração de um fluxograma que descreve o funcionamento da situação proposta. Com base nisso, foi construído o diagrama de classes, definindo os elementos necessários e seus respectivos comportamentos. Por fim, desenvolveu-se o algoritmo responsável por realizar a simulação completa do cenário.

3.1 Fluxograma

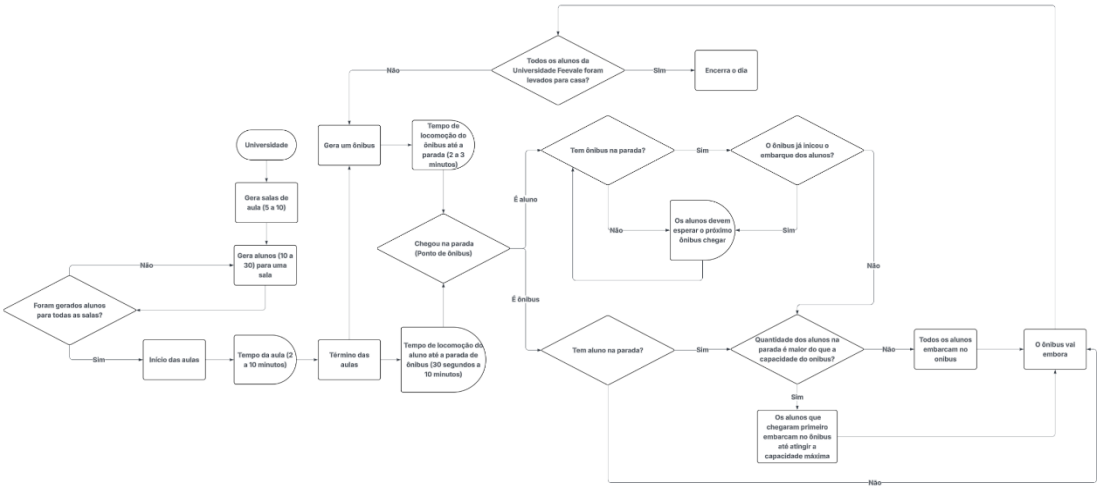

Imagem 1: Fluxograma do projeto. Fonte: Os autores.

Link para visualização do fluxograma: [Fluxograma]

3.2 Diagrama de Classes

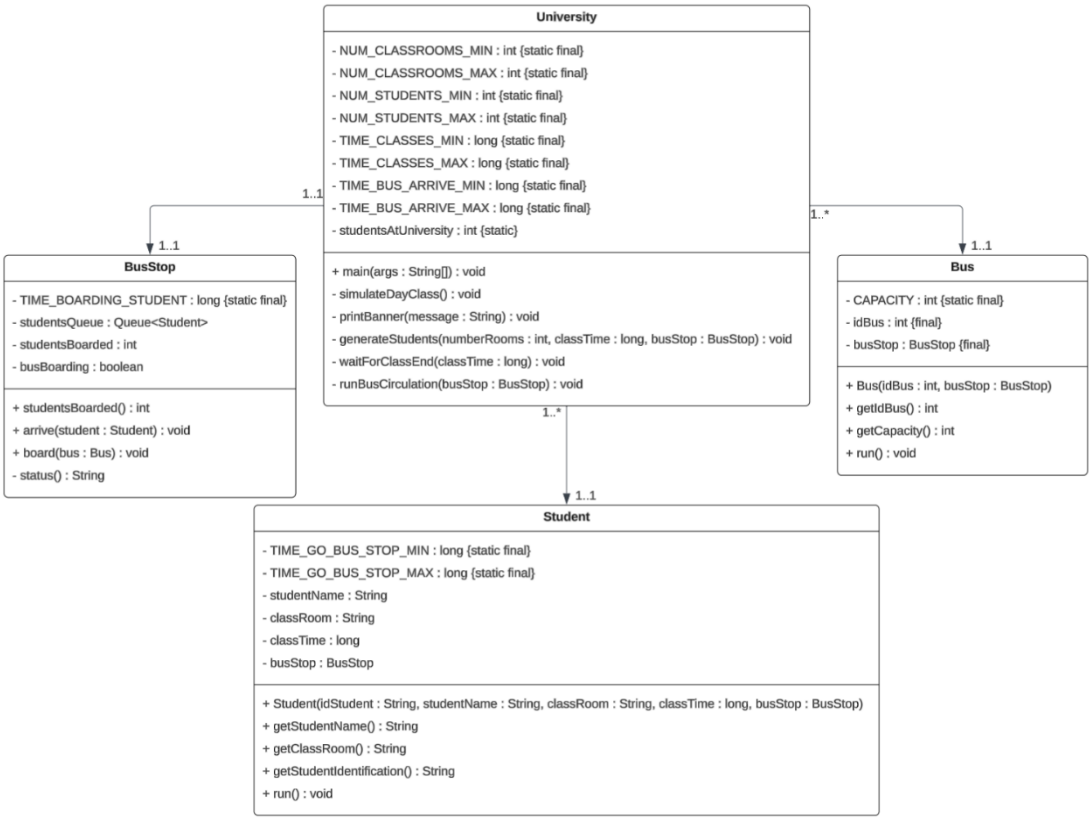

Imagem 2: Diagrama de classes do projeto. Fonte: Os autores.

Link para visualização do diagrama de classes: [Digrama de Classes]

3.3 Linguagem de Programação

O projeto bus-thread foi todo desenvolvido em linguagem Java, organizado no pacote com.feevale, e dividido nas seguintes classes principais:

Classe	Função
University	Classe principal, responsável por inicializar a simulação do dia de aulas dos alunos. Cria salas, alunos, ônibus e gerencia o tempo da simulação.
BusStop	Classe que representa o ponto de ônibus. Controla a fila de espera dos alunos, embarque e sincronização.
Bus	Classe que estende Thread e que representa cada ônibus.
Student	Classe que estende Thread e que representa cada aluno.

3.4 Descrição e Apresentação dos Algoritmos

3.4.1 Classe University.java

```java
/**
 * Gera todas as salas de aula da Universidade Feevale e seus alunos
 *
 * @param numberRooms Número de salas de aula
 * @param classTime   Tempo para as aulas
 * @param busStop     Parada de ônibus da Universidade Feevale
 */
private static void generateStudents(int numberRooms, long classTime, BusStop busStop) {
    // Gera, para cada sala, um número aleatório de alunos
    for (int room = 1; room <= numberRooms; room++) {
        int studentsInRoom = RandomUtils.randomInt(NUM_STUDENTS_MIN, NUM_STUDENTS_MAX);
        System.out.println("Sala " + room + " terá " + studentsInRoom + " alunos.");

        // Cria e dispara a thread para cada um dos alunos
        for (int studentInRoom = 1; studentInRoom <= studentsInRoom; studentInRoom++) {
            String idStudent = "S" + room + "-A" + studentInRoom;
            String studentName;
            if (studentInRoom < 10) {
                studentName = "Aluno-0" + studentInRoom;
            } else {
                studentName = "Aluno-" + studentInRoom;
            }
            String classRoom;
            if (numberRooms < 10) {
                classRoom = "Sala 0" + room;
            } else {
                classRoom = "Sala " + room;
            }
            Student student = new Student(idStudent,
                    studentName,
                    classRoom,
                    classTime,
                    busStop);
            student.start();
            studentsAtUniversity++;
        }
    }

    System.out.println("Total de alunos na universidade Feevale: " + studentsAtUniversity);
}
```

Imagem 3: Método generateStudents da classe University.java. Fonte: Os autores.

O método *generateStudents* é responsável pela criação dos alunos da Universidade Feevale. Nele é executado um laço das salas de aula gerando os alunos correspondentes a cada uma delas. Nesse ponto já é disparada a thread do aluno criado, através do método *start()*.

```java
/**
 * Gera todos os ônibus até que não haja mais alunos na Universidade Feevale
 *
 * @param busStop Parada de ônibus da Universidade Feevale
 */
private static void runBusCirculation(BusStop busStop) throws InterruptedException {
    System.out.println(">>> Circulação de ônibus iniciada...");

    List<Bus> busList = new ArrayList<>();
    int idBus = 1;

    while (true) {
        // Gera um novo ônibus
        Bus bus = new Bus(idBus++, busStop);

        // Aguarda o tempo até a chegada do ônibus
        Thread.sleep(RandomUtils.randomMillis(TIME_BUS_ARRIVE_MIN, TIME_BUS_ARRIVE_MAX));

        // Se todos os alunos na Universidade Feevale já embarcaram
        if (busStop.studentsBoarded() >= studentsAtUniversity) {
            break;
        }

        // Dispara a thread do ônibus
        // Obs.: Realiza o embarque dos alunos que já estão na fila
        bus.start();
        busList.add(bus);
    }

    // Espera todos os ônibus terminarem
    for (Bus bus : busList) {
        bus.join();
    }

    System.out.println(">>> Circulação de ônibus encerrada...");
}
```

Imagem 4: Método runBusCirculation da classe University.java. Fonte: Os autores.

O método *runBusCirculation* é um dois mais importantes da classe *University.java*, pois é nele que é feita a geração dos ônibus conforme a necessidade. Após o final das aulas, esse método é chamado. A cada iteração do *while* é aguardado um tempo aleatório entre 2 e 3 minutos e então testado se os estudantes que já embarcaram (considerando todos os ônibus que já passaram) é maior que o número de alunos da Universidade Feevale. A ideia desse teste é que, se já embarcaram todos os alunos, não precisa ser mais gerado nenhum ônibus e assim o laço se encerra, terminando de gerar os ônibus. O importante de ressaltar

é que cada ônibus é adicionado a uma lista, para que, no final, seja feito o *join()* das threads, aguardando elas encerrar para só assim seguir na execução.

3.4.2 Classe Student.java

```
@Override
public void run() {
    try {
        // Aguarda o tempo do aluno na aula
        Thread.sleep(classTime);

        // Aguarda o tempo que o aluno leva até o ponto de ônibus (aleatório)
        Thread.sleep(RandomUtils.randomMillis(TIME_GO_BUS_STOP_MIN, TIME_GO_BUS_STOP_MAX));

        synchronized (this) {
            // Indica a chegada do aluno no ponto de ônibus
            busStop.arrive(this);

            // Aguarda ser notificado (embarque no ônibus)
            this.wait();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Imagem 5: Método run da classe Student.java. Fonte: Os autores.

O método *run* da classe *Student.java* é o principal método dessa classe. É ele que é chamado quando é feito o *start* de um objeto do tipo *Student*. O *run* é o método de início da thread. Quando é feito o *start* de um *Student*, primeiramente aquele aluno aguarda o tempo da aula recebido da classe *University.java*. Na sequência é aguardado um tempo aleatório que simula o tempo que o aluno leva até a parada de ônibus. Após esse tempo é indicada a chegada do aluno na parada de ônibus através do *busStop.arrive(this)*. Nesse momento é importante destacar o *synchronized* feito que tem o intuito de sincronizar a chegada dos alunos, mantendo a ordem em que a thread (aluno) chegou efetivamente, sem correr risco de furar filas ou desconsiderar algum aluno da contagem. Também se faz o uso do método *wait()* que faz com que aquela thread pare até receber um alerta de que pode voltar sua execução. A ideia desse *wait()* e fazer com que a thread do aluno só se encerre efetivamente quando o aluno embarcar no ônibus, já que no momento do *arrive* ele somente chegou na parada. No método *board* da classe *BusStop.java* é feito o *notify()* para o aluno no momento que aquele aluno embarcou efetivamente. Sem o

wait() e o *notify()*, a thread do aluno seria encerrada sempre quando ele chegar na parada de ônibus, ficando inconsistente com a realidade.

3.4.3 Classe BusStop.java

```java
// Chegada do aluno na fila para pegar o ônibus
public void arrive(Student student) {
    synchronized (studentsQueue) {
        studentsQueue.add(student);

        String message = student.getStudentIdentification() + " chegou no ponto";
        if (busBoarding) {
            message += " durante o embarque e vai esperar o próximo ônibus";
        }
        System.out.println(message + ". " + status());
    }
}
```

Imagem 6: Método arrive da classe BusStop.java. Fonte: Os autores.

```java
// Chegada do ônibus na parada e embarque de alunos até a capacidade máxima
public void board(Bus bus) {
    System.out.println("Ônibus " + bus.getIdBus() + " chegou no ponto.");

    // Capacidade máxima do ônibus
    int capacity = bus.getCapacity();
    // Quantidade de alunos na parada no momento da chegada do ônibus
    int studentsAtBusStop;
    // Número total de alunos que embarcaram no ônibus atual
    int boarded = 0;

    synchronized (studentsQueue) {
        studentsAtBusStop = Math.min(studentsQueue.size(), capacity);
        busBoarding = true;
    }

    // Realiza o embarque dos alunos que já estavam na parada quando o ônibus chegou
    for (int studentAtBusStop = 0; studentAtBusStop < studentsAtBusStop; studentAtBusStop++) {
        Student student;

        synchronized (studentsQueue) {
            student = studentsQueue.poll();
            studentsBoarded++;
            boarded++;

            System.out.println("Ônibus " + bus.getIdBus() + ": " + student.getStudentIdentification()
                    + " embarcou. " + status());
        }

        try {
            // Simula o tempo de embarque do aluno
            Thread.sleep(TIME_BOARDING_STUDENT);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Notifica que o aluno embarcou
        synchronized (student) {
            student.notify();
        }
    }

    synchronized (studentsQueue) {
        busBoarding = false;
    }

    // Se nenhum aluno embarcou
    if (boarded == 0) {
        System.out.println("Ônibus " + bus.getIdBus() + " partiu vazio.");
    } else {
        System.out.println("Ônibus " + bus.getIdBus() + " partiu com " + boarded + " alunos.");
    }
}
```

Imagem 7: Método board da classe BusStop.java. Fonte: Os autores.

O método *arrive* e o *board* da classe *BusStop.java* são os principais métodos dessa classe. O método *arrive* faz a sincronização com a lista de alunos, controlando quando o aluno chegou no ponto e se chegou durante o embarque do ônibus ou não. A sincronização é realizada diretamente na lista de alunos para

tornar possível o método arrive ser executado durante a execução do método *board*. Dessa forma, o método *board* só irá bloquear a chamada do método *arrive* quando estiver executando um bloco de código que também está sincronizado com a lista de alunos, como por exemplo no momento em que ele vai retornar a quantidade de alunos na parada no momento em que o ônibus chega, isso é necessário porquê, usando a sincronização nesse ponto, conseguimos garantir que enquanto ele está retornando essa informação, o método *arrive* não poderá ser executado, trazendo segurança na nossa aplicação. O outro momento seria quando removemos o aluno da lista de alunos e exibimos a mensagem que ele embarcou, também é importante haver sincronização nesse ponto para garantir a consistência dessas informações no nosso código. A sincronização na lista de alunos também é fundamental para exibirmos quando o aluno chegou durante o embarque. Podemos visualizar ali no código que possuímos um tempo de embarque, definimos 1 segundo como o tempo para o aluno realizar o embarque, é crucial perceber que esse tempo não está dentro da sincronização, isso nos permite, durante esse tempo, executar o método *arrive*, tornando possível esse controle de alunos chegando no ponto durante o embarque do ônibus.

3.5 Execução da Aplicação

A execução da aplicação é muito simples, sendo necessário apenas executar a aplicação (apertar o “run”) na classe *University.java*. As saídas da aplicação são exibidas diretamente via terminal do VSCode.

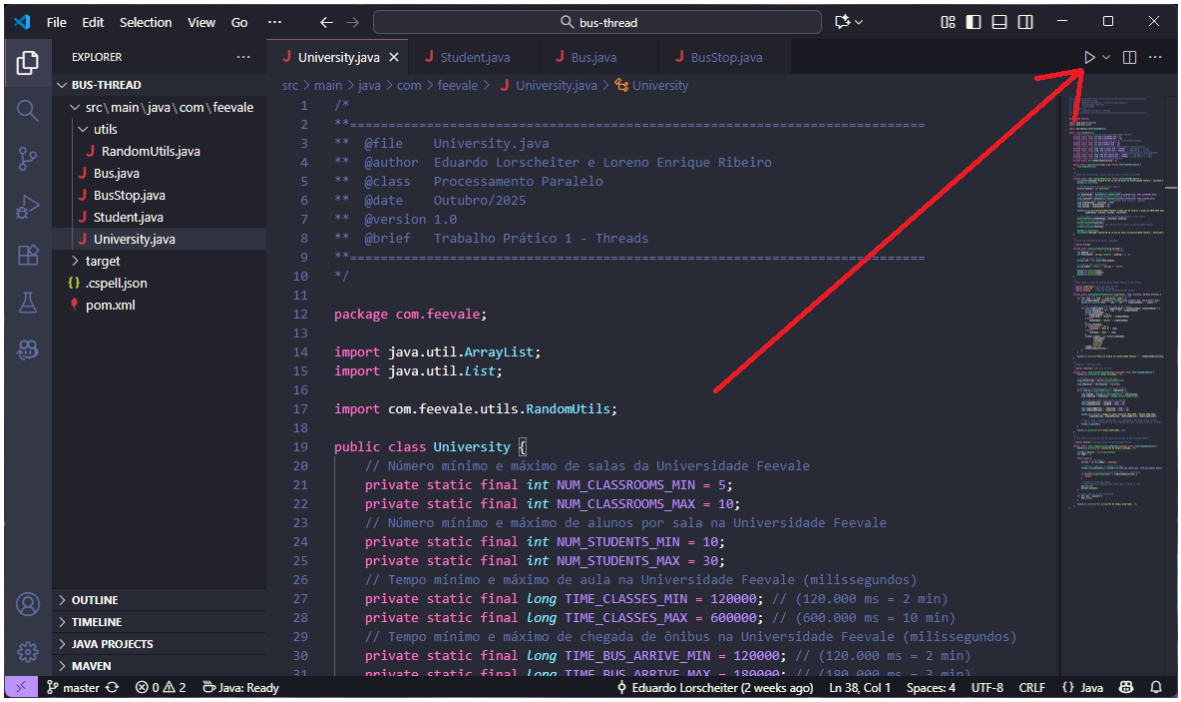

Imagem 8: Executar o projeto. Fonte: Os autores.

4. Exemplo de Aplicação Real

O problema do ônibus apresentado neste trabalho é um exemplo clássico de sincronização em sistemas concorrentes, frequentemente utilizado para ilustrar condições de corrida, exclusão mútua e coordenação entre threads.

Na simulação desenvolvida, o ponto de ônibus representa um recurso compartilhado, enquanto os alunos e ônibus são processos independentes que precisam acessá-lo de forma ordenada. Essa situação reflete diversos problemas reais da computação paralela, como o controle de acesso a regiões críticas e a necessidade de evitar que múltiplos processos modifiquem simultaneamente o mesmo recurso.

Um exemplo prático da aplicação desse tipo de problema ocorre em servidores web: múltiplos usuários (como os alunos) fazem requisições

simultâneas, e o servidor (como o ônibus) deve processar um número limitado de conexões por vez, mantendo as demais em espera até que haja disponibilidade.

5. Conclusão

A implementação do projeto bus-thread permitiu compreender de forma prática os principais conceitos de processamento paralelo e sincronização de threads.

Durante o desenvolvimento, foi possível observar a importância do controle de acesso aos recursos compartilhados e da utilização correta de estruturas de sincronização, como o uso de *synchronized*, *wait()* e *notify().*

A simulação evidenciou que a execução paralela de tarefas melhora o desempenho e reflete o comportamento real de sistemas que operam simultaneamente. Além disso, o uso de threads trouxe maior realismo à dinâmica entre alunos e ônibus, demonstrando o poder e a flexibilidade do paradigma concorrente na programação moderna.

Em resumo, o projeto proporcionou uma visão clara da importância das threads na otimização de aplicações e no controle eficiente de processos que ocorrem ao mesmo tempo.

5.1 Projeto no GitHub

O versionamento do projeto foi todo feito via GitHub. Para maiores consultas pode ser acessado o repositório no seguinte link: Projeto bus-thread

5.2 Vídeo de Explicação do Projeto

Foi disponibilizado um vídeo no YouTube com a explicação do código e do funcionamento do projeto Bus-Thread. O vídeo pode ser visualizado no seguinte link: Vídeo bus-thread