Universidade Estadual Do Ceará

Ciência da computação

CCT – Centro de Ciência e Tecnologia

Relatório do problema do Controle de Tráfego Aéreo

Aluno: Anderson Bezerra Ribeiro

Matrícula: 1196151

Disciplina: Programação Paralela e Concorrente - 2016.2

Introdução

Este relatório apresentará o problema do Controle de Tráfego Aéreo, os conceitos envolvidos e a solução para o mesmo, detalhando a implementação e execução da solução. Por fim serão apresentados os resultados obtidos e comentários sobre os problemas encontrados durante o desenvolvimento.

Problema do Controle de Tráfego Aéreo

O problema trata de um aeroporto que possui somente uma pista, recurso este compartilhado entre aviões que irão pousar e decolar, que deve ser gerenciada para que não haja conflito entre os aviões. A cada n segundos um avião é gerado aleatoriamente no ar ou no aeroporto. Os aviões criados são inseridos em uma fila de tamanho limitado e devem ser gerenciados para que sempre haja espaço para colocar um avião recém-criado.

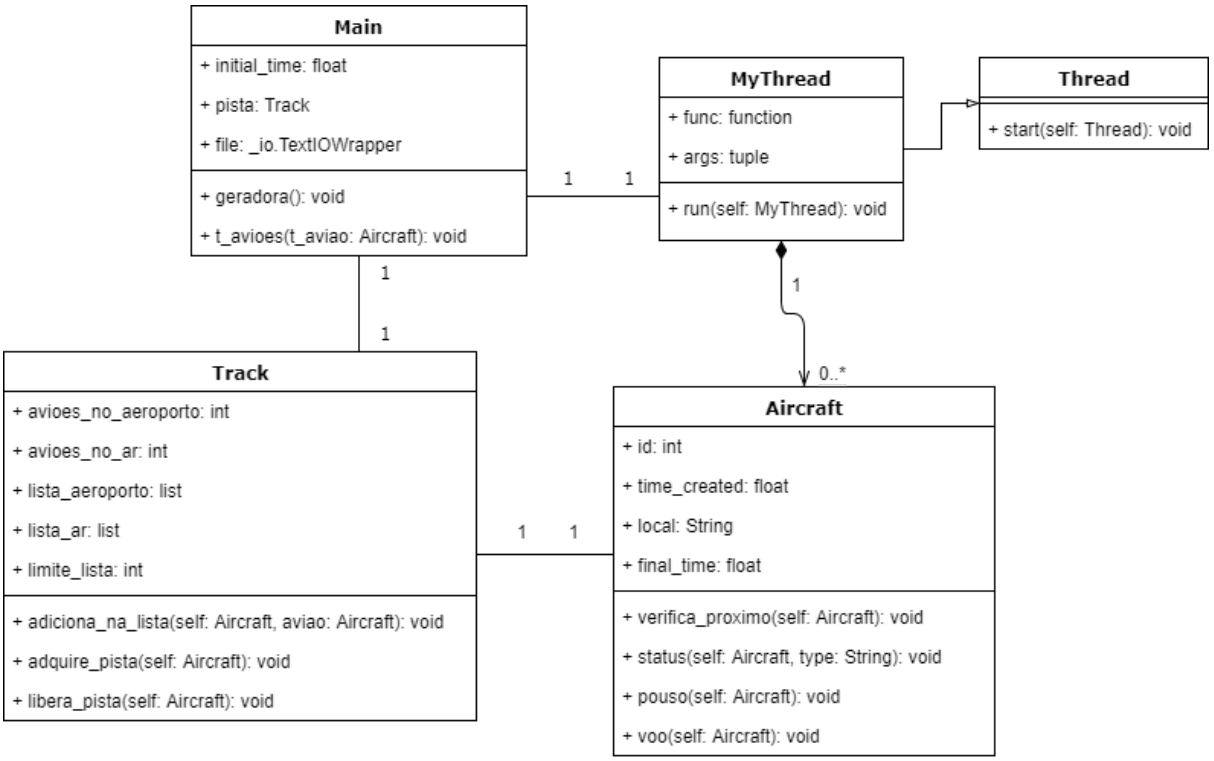
O problema possui algumas especificações e restrições, como descritas a seguir:

- A fila de aviões é limitada, como citado anteriormente, e pequena. Nesta implementação usaremos uma fila de tamanho 4;
- Os aviões são criados em intervalos de tempo fixos. Utilizaremos um intervalo de 8 segundos.
- Um avião no ar possui combustível limitado. Dessa forma, não poderá passar mais que 30 segundos no ar após ser criado;
- Antes de decolar, o avião deve esperar 5 segundos para que o avião anterior que aterrissou saia da pista principal ou o avião que acabou de decolar se afaste com segurança;
- Aeronaves que aterrissam precisam de 10 segundos, também durante o qual nenhuma outra aeronave pode decolar ou pousar;
- A execução termina quando todos os aviões são criados e despachados. Criaremos 10 aviões no ar e 10 no aeroporto.

Descrição da Implementação

A solução foi implementada na linguagem Python, utilizando a IDE PyCharm, da JetBrains. Para implementar a concorrência, criando uma thread para cada avião do programa, e a sincronização entre threads, utilizando semáforos, foi utilizado a biblioteca nativa da linguagem “threading”, que fornece implementações de alto nível desses conceitos. Para acessar os semáforos, foram utilizadas as funções da classe “Semaphore”, “acquire” e “release”, para adquirir e liberar o semáforo, e as funções da classe “Event”, “wait”, “set” e “clear”, para fazer setar uma flag que informará se é a vez do avião ou não.

O diagrama de classes do projeto está descrito a seguir:



- **Main:** Classe principal do programa. Possui as variáveis de contagem de tempo, pista e de arquivo. A função que define o comportamento da thread geradora de aviões está declarada aqui, juntamente com a função que define o comportamento de cada thread de avião.
- **MyThread:** Classe que herda os métodos de criação da classe nativa “Thread”. Ela cria e inicia a execução de uma thread, definindo uma função base e uma tupla de argumentos para a mesma.
- **Thread:** Classe nativa da linguagem para criação de threads.
- **Aircraft:** Possui os atributos de tempo inicial, local de criação e tempo final de cada avião. Os método dessa função são executados por cada thread avião. O primeiro é executado repetidamente até que o avião reconheça que é sua vez, caso não seja, ela entra em espera até que a pista seja liberada. O método status imprime na tela o registro da última ação do avião. Também possui os métodos de pouso e vôo, onde um deles será executado pelo

avião com prioridade. Esse método bloqueia a pista até que ele a use por completo e depois alerta as outras threads em espera.

- **Track:** Essa é a classe que é executada concorrentemente pelas outras threads, o recurso compartilhado. Ela possui os atributos de contagem de aviões, quantos já foram criados no ar e no aeroporto, e os atributos de listas, lista de aviões em cada fila e o limite delas. Ela possui um método para adicionar um avião criado em sua respectiva fila e dois métodos de controle de semáforo.

Descrição dos algoritmos

Ao ser criada, cada thread executa o método base “t_avioes”. Este é um método sincronizado, que coloca a thread em espera cada vez que ela vê que não é a próxima a utilizar a pista, utilizando um “wait””.

```
def t_avioes(t_aviao):
    '''Função base da thread do avião que o coloca em loop até que seja sua vez'''
    if t_aviao.local == "Aeroporto":
        while t_aviao in pista.lista_aeroporto:
            if not pista.turn.is_set():
                pista.turn.wait()
            t_aviao.verifica_proximo()
    else:
        while t_aviao in pista.lista_ar:
            if not pista.turn.is_set():
                pista.turn.wait()
            t_aviao.verifica_proximo()
```

Quando a pista termina de ser usada, um comando “set” acorda todas as threads e elas entram na função “verifica_proximo” para avaliar se é sua vez. Caso não seja, voltam a dormir.

As condições para que um avião no ar utilize a pista são:

- Ser o primeiro da fila de aviões do ar;
- Estar em tempo crítico (mais de 20 segundos voando) ou ter mais aviões no ar do que no aeroporto.

As condições para que um avião no aeroporto utilize a pista são:

- Ser o primeiro da fila de aviões no aeroporto;
- Não ter aviões no ar ou, se houver, não ter aviões em tempo crítico.

Se alguma das condições for atendida, a variável turn é setada para 0 com a função “clear”, garantindo que as outras threads durmam ao executar o “wait”, depois o avião chama a função de pouso/voo.

```python
@synchronized
def verifica_proximo(self):
    '''Verifica se o avião é o próximo a usar a pista. Se não for, entra em espera'''
    if pista.mutex_pista._value > 0:
        # Verifica previamente o valor do mutex para evitar aleatoriedade na escolha
        if pista.lista_ar and pista.lista_ar[-1] == self:  # Verifica se é a sua vez
            tempo = time() - initial_time - self.time_created
            if tempo > 20:
                # Se avião tiver sido criado a mais de 20 segundos
                pista.turn.clear()
                self.pouso()
                return
            elif len(pista.lista_ar) > len(pista.lista_aeroporto):
                # Se o número de aviões no ar for maior ou igual aos do aeroporto
                pista.turn.clear()
                self.pouso()
                return
        elif pista.lista_aeroporto and pista.lista_aeroporto[-1] == self:
            if len(pista.lista_ar) <= len(pista.lista_aeroporto):
                if not pista.lista_ar:
                    # Se não houver aviões no ar
                    pista.turn.clear()
                    self.voo()
                    return
                elif pista.lista_ar and (time() - initial_time - pista.lista_ar[-1].time_created) < 20:
                    # Se houver mais aviões no aeroporto e nenhum criado a no mínimo 20 segundos no ar
                    pista.turn.clear()
                    self.voo()
                    return
```

As funções de pouso e vôo são semelhantes, mudando somente o local em que o avião se encontra. Ao iniciar a função uma mensagem avisando que a pista está sendo usada é impressa e logo em seguida a bloqueia. Depois, a lista de aviões é bloqueada para que o avião possa ser removido de forma segura. Após a remoção o tempo final é definido e uma mensagem de status é impressa. A lista é então liberada e a thread dorme por 10 segundo, tempo referente a duração de pouso ou decolagem. Por fim, a pista é liberada, o log é escrito no arquivo, a mensagem de liberação é impressa e as outras threads são acordadas, reiniciando o ciclo.

```
def pouso(self):
    '''Bloqueia a pista, realiza o pouso e libera a pista'''
    print("-" * 50 + "\n{:.4f}s - <AVIÃO {}> OCUPANDO A PISTA PARA POUSO\n".format(time() - initial_time, self.id) + "-" * 50)

    pista.adquire_pista()
    pista.mutex_ar.acquire()
    self.final_time = (time() - initial_time)
    pista.lista_ar.pop()
    self.status("remover")
    pista.mutex_ar.release()
    sleep(10)
    pista.libera_pista()

    file.write('-' + self.__str__() + '\n')
    print("-" * 50 + "\n{:.4f}s - <AVIÃO {}> LIBEROU A PISTA\n".format(time() - initial_time, self.id) + "-" * 50)

    pista.turn.set()
```

```
def voo(self):
    '''Bloqueia a pista, realiza o voo e libera a pista'''
    print("-" * 50 + "\n{:.4f}s - <AVIÃO {}> OCUPANDO A PISTA PARA VOO\n".format(time() - initial_time, self.id) + "-" * 50)

    pista.adquire_pista()
    pista.mutex_aeroporto.acquire()
    self.final_time = (time() - initial_time)
    pista.lista_aeroporto.pop()
    self.status("remover")
    pista.mutex_aeroporto.release()
    sleep(10)
    pista.libera_pista()

    file.write('-' + self.__str__() + '\n')
    print("-" * 50 + "\n{:.4f}s - <AVIÃO {}> LIBEROU A PISTA\n".format(time() - initial_time, self.id) + "-" * 50)

    pista.turn.set()
```

Execução

```
0.0165s - <Avião 1> gerado no Ar - Fila Ar/Aeroporto [1]/[0] - Total Ar/Aeroporto: 1/0
-------------------------------------------------
0.0391s - <AVIÃO 1> OCUPANDO A PISTA PARA POUSO
-------------------------------------------------
0.0552s - <Avião 1> removido do Ar - Fila Ar/Aeroporto [0]/[0] - Total Ar/Aeroporto: 1/0
8.0450s - <Avião 2> gerado no Ar - Fila Ar/Aeroporto [1]/[0] - Total Ar/Aeroporto: 2/0
-------------------------------------------------
10.0678s - <AVIÃO 1> LIBEROU A PISTA
-------------------------------------------------
-------------------------------------------------
10.0788s - <AVIÃO 2> OCUPANDO A PISTA PARA POUSO
-------------------------------------------------
10.0793s - <Avião 2> removido do Ar - Fila Ar/Aeroporto [0]/[0] - Total Ar/Aeroporto: 2/0
16.0608s - <Avião 3> gerado no Aeroporto - Fila Ar/Aeroporto [0]/[1] - Total Ar/Aeroporto: 2/1
-------------------------------------------------
20.0847s - <AVIÃO 2> LIBEROU A PISTA
-------------------------------------------------
-------------------------------------------------
20.0897s - <AVIÃO 3> OCUPANDO A PISTA PARA VOO
-------------------------------------------------
20.0997s - <Avião 3> removido do Aeroporto - Fila Ar/Aeroporto [0]/[0] - Total Ar/Aeroporto: 2/1
24.0818s - <Avião 4> gerado no Aeroporto - Fila Ar/Aeroporto [0]/[1] - Total Ar/Aeroporto: 2/2
-------------------------------------------------
30.1099s - <AVIÃO 3> LIBEROU A PISTA
-------------------------------------------------
```

A imagem acima mostra o início da execução do programa. Nela podemos ver o momento em que cada avião é criado ou removido e o momento em que a pista é ocupada ou liberada.

Abaixo, a imagem mostra o fim da execução. Quando todos os 20 aviões foram criados e despachados.

```
-------------------------------------------------
190.4378s - <AVIÃO 20> OCUPANDO A PISTA PARA VOO
-------------------------------------------------
190.4478s - <Avião 20> removido do Aeroporto - Fila Ar/Aeroporto [0]/[0] - Total Ar/Aeroporto: 10/10
-------------------------------------------------
200.4480s - <AVIÃO 20> LIBEROU A PISTA
-------------------------------------------------

200.4480: Fim da execução

Nenhum avião foi derrubado durante o desenvolvimento desse programa
```

Resultados

A implementação do trabalho prático ajudou a consolidar os conceitos vistos nas aulas teóricas. Pude entender melhor o conceito de deadlock e verificar, por experiência própria, os casos que resultam nele. Reforçar e utilizar os conceitos de métodos protegidos e sincronização foram a parte que considerei mais importante, visto que todo o projeto gira em torno de sincronização. Por fim, posso dizer que o trabalho foi de suma importância para a conclusão da disciplina.