

Relatório:

Trabalho II - Winter Park

Nicolas Elias e Maurício Konrath

Universidade Federal de Santa Catarina

Curso de Bacharelado em Ciências da Computação

Disciplina INE5410 - Programação Concorrente

Prof. Frank Augusto Siqueira

Prof. Márcio Bastos Castro

Sobre a elaboração do trabalho: Foi utilizado o LiveShare disponível no Visual Studio Code e o Discord para fazermos chamadas, dessa forma os dois participavam e podiam editar em simultâneo.

Descreva as estratégias utilizadas na sua implementação para controlar o acesso aos seguintes recursos:

0) equipamento de proteção:

Primeiro de tudo antes de qualquer coisa, o cliente que entra no parque pega um equipamento de proteção:

```
# Cliente pega um kit com os equipamentos de proteção
def pegar Equip_protecao(self):
    # FUNCIONARIO 0 ENTREGA UM EQUIPAMENTO DE PROTEÇÃO
    self.equip_atual = init.funcionarios[0].entrega_equipamento()

    self.log("Pegou um kit com equipamentos de proteção.")
```

O funcionário 0 que é o responsável pelo equipamento de proteção é chamado e retorna um equipamento. Como todos os equipamentos têm um número máximo, foi implementado uma lógica no arquivo equipamentos.py que faz uma fila para receber esse equipamento caso todos estejam sendo utilizados:

```
class Equipamentos:
    """
    Equipamentos representa um conjunto de equipamentos de um determinado tipo.
    Você deve implementar os métodos que controlam a entrega e devolução de
    equipamentos, respeitando as restrições impostas no enunciado do trabalho.
    """
    # Construtor da classe que representa um conjunto de equipamentos
    def __init__(self, nome_equipamento, quant_equipamentos):
        self.nome = nome_equipamento
        self.quantidade = quant_equipamentos
        self.disponibilidade_equip = Semaphore(self.quantidade)

    def pegar_equipamento(self, equip):
        """
        IMPLEMENTE AQUI: Entrega de um equipamento
        """
        self.disponibilidade_equip.acquire()

        self.quantidade -= 1

    def devolver_equipamento(self):
        """
        IMPLEMENTE AQUI: Devolução de um equipamento
        """
        # LIBERA UMA VAGA

        self.disponibilidade_equip.release()
        self.quantidade += 1
```

Semáforo iniciado com a quantidade de tal equipamento, sempre que um funcionário pega equipamento, ocupa uma vaga, e quando devolve, libera uma vaga.

```
# Cliente devolve um kit com os equipamentos de proteção
def devolver Equip_protecao(self):
    # FUNCIONARIO 0 RECEBE UM EQUIPAMENTO DE PROTEÇÃO
    init.funcionarios[0].recebe_equipamento(self.equip_atual)
    init.sem_limpezas[0].release()
    self.log("Devolveu um kit com equipamentos de proteção.")
```

No método de devolver segue a mesma lógica do de pegar, entrega a um funcionário que por sua vez, devolve um equipamento liberando uma vaga da fila de equipamentos, a única diferença é que tem um release em um semáforo que serve como uma flag, para esse mesmo funcionário limpar o equipamento devolvido.

a) pista de patinação:

Primeiro o cliente pega do funcionário encarregado dos patins um equipamento de patinação:

```
# Cliente pega um par de patins para usar a pista de patinação
def pegar_patins(self):
    # FUNCIONARIO 1 ENTREGA EQUIPAMENTO DE PATINS
    self.equip_atual = init.funcionarios[1].entrega_equipamento()
    self.log("Pegou um par de patins.")
```

Armazenado em um atributo equip_atual.

Depois de pegar o equipamento de patinação, o cliente vai para o método de aguardar pista:

```
# Cliente aguarda que haja lugar na pista de patinação
def aguardar_lugar_pista(self):
    # CLIENTE AGUARDA UM LUGAR NA PISTA DE PATINAÇÃO DA SEGUINTE FORMA:
    # CHAMA O METODO DA ATRACAO 0 = PATINAÇÃO
    # METODO ENTRAR NA ATRACAO TEM UM SEMAFORO QUE CONTA QUANTOS CLIENTES É PERMITIDO
    # OCUPA UMA DAS VAGAS DA CAPACIDADE DA ATRACAO
    init.atracoes[0].entrar_atracao()
    self.log("Entrou na pista de patinação.")
```

Que por sua vez tem o comportamento de chamar o método da atração 0 de entrar na atração (atração 0 é a pista de patinação).

Dentro do método de entrar na atração possui a seguinte estrutura:

```

# Construtor da classe que representa uma atração do parque
def __init__(self, nome_atracao, capacidade_atracao):
    self.nome = nome_atracao          # Nome da atração
    self.capacidade = capacidade_atracao # Limite de clientes na atração
    self.disponibilidade_atracao = Semaphore(capacidade_atracao)

def entrar_atracao(self):
    """
    IMPLEMENTE AQUI: Entrada do cliente na atração
    """
    self.disponibilidade_atracao.acquire()

def sair_atracao(self):
    """
    IMPLEMENTE AQUI: Saída do cliente da atracão
    """
    #LIBERA UMA VAGA
    self.disponibilidade_atracao.release()

```

Que possui um semáforo que é iniciado com a quantidade de vagas que a atração possui, então sempre que o método entrar na atração é chamado ele toma uma das vagas para si.

Após entrar o método patinar é chamado no run, e quando acaba o tempo da atração, o método sair atração é chamado:

```

def patinar(self):
    self.log("Está patinando.")
    self.tempo_atracao()
    # APOS PATINAR (TEMPO DA ATRAÇÃO) CLIENTE CHAMA METODO DA ATRAÇÃO
    # QUE ELE SE ENCONTRA LIBERANDO UMA VAGA PARA PATINAR
    # SEMAFORO PRESENTE NO ARQUIVO ATRACOES.PY
    init.atracoes[0].sair_atracao()

```

E o método de sair da atração libera uma das vagas que a atração patinar possuía.

Após sair da atração, o cliente deve devolver seus patins então o método é chamado no run, devolver patins:

```

# Cliente devolver os patins que estava usando
def devolver_patins(self):
    # FUNCIONARIO 1 RECEBE UM EQUIPAMENTO DE PATINAÇÃO
    init.funcionarios[1].recebe Equipamento(self.equip_atual)
    init.sem_limpezas[1].release()
    self.log("Devolveu um par de patins.")

```

O funcionário encarregado dos patins recebe o equipamento atual do cliente no caso o patins, e uma flag que consiste em um semáforo binário libera a limpeza do equipamento recebido pelo funcionário, `init.sem_limpezas` consiste em uma lista de semáforos para cada

funcionário, para ser limpo de forma independente (Será melhor explicado na area do funcionario).

b) teleférico;

O teleférico como a pista de patins, não tem muitas complicações, é primeiramente chamado o método pegar teleférico:

```
# Cliente aguarda um lugar no teleférico
def pegar_teleferico(self):
    # SEGUE O RACIOCINIO DESCRITO NOS COMENTARIOS DO METODO def aguardar_lugar_pista(self):
    init.atracoes[1].entrar_atracao()
    self.log("Pegou cadeira no teleférico.")
    # APENAS MARCAM TEMPO DE SUBIDA E DESCIDA OS METODOS ABAIXO
```

Ele se comporta da seguinte forma: o método da atração 1 entrar atração é chamado, que contém os semáforos de controle de entrada, como cadeiras são individuais cada vez que alguém entra uma das cadeiras (vagas) são ocupadas.

Os métodos subir e descer, contam apenas como medidas de tempo que leva para descer e subir, o método run decide quais serão chamados de forma aleatória.

c) pista de esqui/snowboard;

O Funcionário entrega para o cliente um esqui ou snowboard:

```
# Cliente pega esquis
def pegar_esquis(self):
    # PEGA DO FUNCIONARIO 2 UM EQUIP. DE ESQUIS
    self.equip_atual = init.funcionarios[2].entrega Equipamento()
    self.log("Pegou esquis.")
```

```
# Cliente pega uma prancha de snowboard
def pegar_snowboard(self):
    # FUNCIONARIO 3 ENTREGA UM EQUIP. DE SNOWBOARD
    self.equip_atual = init.funcionarios[3].entrega Equipamento()
    self.log("Pegou um snowboard.")
```

Nessa lógica já explicada, o funcionário resgata um equipamento e retorna para o cliente.

Já a parte de aguardar diferente dos patins e teleférico, à montanha é utilizada por duas atrações, então a fila é compartilhada entre eles, a forma que pensamos nos patins já supre essa necessidade por se tratar de um semáforo de vagas, cada um deles vai ocupando uma vaga até não ter mais, próximos que tentarem obter entraram na fila do semáforo até alguém sair da atração da montanha sul

```
# Cliente desce a montanha esquiando
def descer_esquiando(self):
    self.log("Começa a descer a montanha esquiando.")
    self.tempo_atracao()
    self.log("Terminou de descer a montanha esquiando.")
    # LIBERA UMA VAGA DA MONTANHA SUL
    init.atracoes[2].sair_atracao()
```

```
# Cliente desce a montanha com uma prancha de snowboard
def descer_snowboard(self):
    self.log("Começou a descer a pista de snowboard.")
    self.tempo_atracao()
    self.log("Desceu a pista de snowboard.")
    # APOS DESCER, CLIENTE SAI DA ATRAÇÃO
    init.atracoes[2].sair_atracao()
```

E aqui está a mesma lógica dos patins para devolução:

```
# Cliente devolve os esquis
def devolver_esquis(self):
    # FUNCIONARIO 2 RECEBE UM EQUIP. DE ESQUI
    init.funcionarios[2].recebe Equipamento(self.equip_atual)
    init.sem_limpezas[2].release()
    self.log("Devolveu os esquis.")
```

```
# Cliente devolve uma prancha de snowboard
def devolver_snowboard(self):
    # FUNCIONARIO 3 RECEBE UM EQUIP DE SNOWBOARD
    init.funcionarios[3].recebe Equipamento(self.equip_atual)
    init.sem_limpezas[3].release()
    self.log("Devolveu um snowboard.")
```

d) pistas de trenó/bobsled;

Aqui se trata da parte que depende de uma grande parte da sincronização presente no trabalho (junto com a parte de limpeza e disponibilidade de funcionários), pois só pode descer 1 dos dois por vez, e o bobsled é utilizado em duplas, então ha um atributo em cada cliente que no método formar duplas, é atribuído para um da dupla que deve pegar um bobsled o outro que formou a dupla não deve pegar nada e apenas descer

```
# Cliente aguarda um bobsled livre para descer a montanha
def pegar_bobsled(self):
    # FUNCIONARIO 4 ENTRA UM EQUIP DE BOBSLED
    if self.pegar_bob_dupla:
        self.equip_atual = init.funcionarios[4].entrega Equipamento()
        self.log("Pegou um bobsled.")
```

```
# Cliente aguarda um trenó livre para descer a montanha
def pegar_treno(self):
    # FUNCIONARIO 5 ENTREGA UM TRENÓ
    self.equip_atual = init.funcionarios[5].entrega Equipamento()
    self.log("Pegou um trenó.")
```

Após pegar o equipamento, há dois caminhos no primeiro é o bobsled, caso o cliente decida ir de bobsled ele precisa achar uma dupla, é chamado o método formar duplas:

```
# Cliente aguarda formação da dupla
def formar_dupla(self):
    # SEMAFORO PARA APENAS DUAS THREADS ENTRAREM NA REGIÃO CRÍTICA
    init.sem_apenas2.acquire()
    # MUTEX PARA SO UMA THREAD ESCREVER
    init.lock.acquire()
    init.contador_duplas += 1

    if init.contador_duplas < 2:
        nice = init.condition_dupla.wait(3)
        if not nice:
            self.log("Desistiu de esperar por uma dupla")
            init.contador_duplas = 0
        else:
            self.log("Formou uma dupla")
            init.contador_duplas = 0
    else:
        # APENAS ESSE CLIENTE PEGA EQUIPAMENTO
        self.pegar_bob_dupla = True
        init.condition_dupla.notify()
    init.lock.release()
    init.sem_apenas2.release()
```

Começamos com um semáforo que permite apenas duas threads participarem da lógica, após isso um lock é obtido e o incremento da variável contadora acontece, a primeira condição será satisfeita pela primeira thread que chegar ($\text{contador} < 2$), que vai liberar o lock inicial e estipular um tempo de espera para ser notificado e formar uma dupla, a segunda thread entra na região crítica incrementa novamente o contador, e a primeira condição não é satisfeita, e cair no else que atribui para essa thread que formou dupla pegar o bobsled e a outra que estava esperando continua em false este atributo, pois não deve pegar, é utilizado um por

dupla, e notifica o condition wait, que por sua vez retorna um valor booleano que checa se ele não é verdadeiro, pois se for o notify o notificou então ele cair no else que forma uma dupla e prosseguem, caso não tenha mais nenhum cliente para notificar essa thread que está aguardando dupla, ela simplesmente acaba o timeout e a condição é satisfeita, pois a variável nice é falsa, que aponta que o cliente desistiu de esperar por uma dupla.

OBS: Itens abaixo já foram explicados nas suas respectivas atrações que os utilizam.

- e) patins;
- f) pranchas de snowboard;
- g) esquis;
- h) trenós;
- i) bobsleds.

Extra j) Funcionário

O funcionário tem como já dito anteriormente uma sincronização de semáforos para limpeza e disponibilidade, o de limpeza consiste em uma lista de semáforos para cada funcionário para que ele trabalhe de forma independente:

```
# Criação dos funcionários
for i in range(1,7):
    # LOCKS DO ARRAY LOCKS_FUNC CADA FUNCIONARIO TEM SEU LOCK COM UM INDEX APARA DAR LOCK EM CADA FUNCIONARIO DE FORMA INDIVIDUAL
    lock_disp = Lock()
    locks_func.append(lock_disp)
    sem_l = Semaphore(0)
    sem_limpezas.append(sem_l)
    func = Funcionario(i,equipamentos[i-1], i-1)
    funcionarios.append(func)
```

Esses semáforos são criados no init.py, pois é utilizado de forma global da seguinte forma:

```
# Comportamento do Funcionario
def run(self):
    """
    NÃO ALTERE A ORDEM DAS CHAMADAS ABAIXO.
    """
    self.log("Iniciando o expediente. Gerenciando equipamento "+self.equipamento.nome)
    self.trabalhando = True

    cont Equip limpos = 0
    while self.trabalhando == True :
        # IF PARA CONTROLAR O SEMAFORO QUE PERMITE A LIMPEZA
        # QUANDO O CONTADOR CHEGA NA QUANTIDADE DE EQUIPS LIMPOS ELE FICA TRAVADO POR SE TRATAR DE
        # UM SEMAFORO BINARIO E NÃO PROSSEGUE COM A EXECUÇÃO DO PROGRAMA
        init.sem_limpezas[self.lock_index].acquire()
        self.limpar_equipamento()
        cont Equip limpos += 1
        if cont Equip limpos >= init.num Equip turno:
            self.descansar()
            cont Equip limpos = 0

    self.log("Terminando o expediente")
```

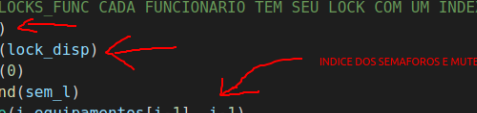

Sempre que o funcionário tentar limpar alguma coisa que não tem, esse semáforo que começa em zero fica travado, até um cliente devolver o respectivo equip de cada funcionário para liberar a thread e limpar o equipamento, no cliente é liberado assim:

```
# Cliente devolve um kit com os equipamentos de proteção
def devolver Equip_protecao(self):
    # FUNCIONARIO 0 RECEBE UM EQUIPAMENTO DE PROTEÇÃO
    init.funcionarios[0].recebe_equipamento(self.equip_atual)
    init.sem_limpezas[0].release()
    self.log("Devolveu um kit com equipamentos de proteção.")
```

Logo devolveu algo, limpa.

A segunda parte é quando o funcionário vai pro descanso, quando ele está descansando ele precisa ficar indisponível pro cliente pegar um equipamento, e como é a thread do cliente que pega o equipamento do funcionário a lógica que utilizamos foi a seguinte:

```
# COLOCANDO FORA DO FOR PARA CONSEGUIR ACESSO EM OUTROS AQUIVOS
# Criação dos funcionários
for i in range(1,7):
    # LOCKS DO ARRAY LOCKS_FUNC CADA FUNCIONARIO TEM SEU LOCK COM UM INDEX APARA DAR LOCK EM CADA FUNCIONARIO DE FORMA INDIVIDUAL
    lock_disp = Lock()
    locks_func.append(lock_disp)
    sem_l = Semaphore(0)
    sem_limpezas.append(sem_l)
    func = Funcionario(i,equipamentos[i-1], i-1)
```



Um lock foi criado para cada funcionário, e um index foi passado como parâmetro, para utilizar nos semáforos e mutexes.

Quando a condição de descanso é satisfeita, é chamado o método descansar:

```
# Funcionário descansa durante um tempo
def descansar(self):
    self.log("Hora do intervalo de descanso.")
    # LOGICA DE TRAVA DE RECEBER E ENTREGAR EQUIPAMENTOS
    init.locks_func[self.lock_index].acquire()

    sleep(init.tempo_descanso * init.unidade_de_tempo)
    self.log("Fim do intervalo de descanso.")

    init.locks_func[self.lock_index].release()
```

Que tem o seguinte comportamento: Exibe que é a hora do descanso do funcionário, obtém seu mutex travando-o faz seu período de descanso e exibe fim do descanso, e libera seu respectivo mutex.

Os clientes que nesse tempo de descanso tentar obter ou devolver algo para o

funcionário que se encontra descansando, acontece a seguinte coisa:

```
# Funcionário entrega um equipamento para um cliente.
def entrega Equipamento(self):
    # CASO O FUNCIONARIO ESTAJA DESCANSANDO ELE FICA PARADO AQUI ATE ACABAR O DESCANSO
    while init.locks_func[self.lock_index].locked():
        pass
    # RESGATA UM EQUIP E O ENTREGA PARA O CLIENTE E RETORNA
    equip = self.equipamento.pegar_equipamento(self.equipamento)
    self.log("Entregou "+self.equipamento.nome+" para um cliente.")

    return equip
```

```
# Funcionário recebe um equipamento.
def recebe_equipamento(self, equip):
    # CASO O FUNCIONARIO ESTEJA NO DESCANSO NÃO PODE RECEBER UM EQUIP, ENTÃO FICA PARADO ESPERANDO O DESCANSO ACABAR
    while init.locks_func[self.lock_index].locked():
        pass
    # DEVOLVE UM EQUIPAMENTO A CONTAGEM DE QTDD EQUIPS
    #init.sem_limpezas[self.lock_index].release()

    self.equipamento.devolver_equipamento()
    self.log("Recebeu "+self.equipamento.nome+" de um cliente.")
```

Enquanto ele estiver com o lock do descanso, a thread que tentou acessar o funcionário no descanso fica esperando ele sair de lá (dando unlock) até que o sleep acabe o funcionário desta forma fica indisponível para os clientes.

Descreva se existe alguma possibilidade de deadlock ou starvation na execução da simulação.

Anteriormente, no método formar duplas tínhamos pensado na seguinte solução:

```

# Cliente aguarda formação da dupla
def formar_dupla(self):
    # SEMAFORO PARA APENAS DUAS THREADS ENTRAREM NA REGIÃO CRÍTICA
    init.sem_apenas2.acquire()
    if init.quant_equip_protecao >= 2 and init.quant_bobsleds >= 2:
        # MUTEX PARA VARIÁVEL GLOBAL
        init.mutex_duplas.acquire()
        # CONTA 1 CLIENTE QUE CHEGOU PARA FORMA DUPLA
        init.contador_duplas += 1
        init.mutex_duplas.release()
        #VERIFICA SE JA TEM DOIS CLIENTES AQUI PARA FORMAR DUPLAS
        if init.contador_duplas == 2:
            #LIBERA OS DOIS CLIENTES PARA DESCER A MONTANHA
            init.contador_duplas = 0
            # APENAS UM EQUIP USADO POR DUPLA, ENTÃO CLIENTE 2 E QUE VEIO FORMAR DUPLA DEVOLVE UM EQUIP
            init.funcionarios[4].recebe Equipamento(self.equip_atual)
            init.sem_limpezas[4].release()
            self.log("Devolveu um bobsled pois formou uma dupla.")
            #LIBERA O PRIMEIRO CLIENTE QUE CHEGOU NO METODO PARA DESCEREM JUNTOS
            init.sem_duplas.release()
            init.sem_duplas.release()
        else:
            #TRAVA O CLIENTE ATE TER UMA DUPLA
            # SE SO TIVER UM CLIENTE SAIR FORA DAQUI
            self.log("Aguardando formar dupla")
            init.sem_duplas.acquire()
            self.log("Formou uma dupla para descer a montanha no bobsled.")
    else:
        self.log("Impossível formar dupla")
    init.sem_apenas2.release()

```

Que funcionava muito bem em praticamente todos os casos, exceto um, o caso que demorou mais pra resolver, que é: quando o último cliente quer formar uma dupla, não tem ninguém para formar e ele ficava no semáforo parado infinitamente esperando alguém, e resolvemos mudar a lógica pois tentar arrumar essa lógica pronta foi uma tarefa cansativa e não conseguimos solucionar; aí pensamos no método descrito na parte de bobsled e trenos. Que funciona de forma muito rápida e semelhante, porém ele abrange a parte de ter uma pessoa só, por conta de um timeout de 3 segundos.

Suponha que, no lugar de threads, fossem usados processos para implementar os clientes e funcionários. Haveria algum ganho no desempenho do programa?

Se trocássemos as threads usadas por processos, poderíamos ter algumas diferenças, por exemplo, em relação ao consumo de memória, onde as threads têm a memória compartilhada (troca de informações muito mais fácil e simples), já os processos requerem uma alocação de memória não compartilhada (utilizando mais recursos que as threads), dificultando a comunicação entre os clientes, os funcionários, os equipamentos e as atrações. A principal desvantagem de utilizar processos, é que seria necessário mecanismos de comunicação para trocar dados. Já a principal vantagem que resultaria em ganhos, seria no paralelismo, em python as threads se revezam entre elas, obtendo um lock global para executar, havendo concorrência entre elas, e não paralelismo real. Os processos executam

paralelamente, sendo um processo para cada tipo do parque, diversos processos clientes obtendo de processos funcionários os equipamentos, e entrando em atrações paralelamente, isso resultaria em um uso maior de recursos, porém com ganhos significativos relacionados ao paralelismo.