Questão 1 .....

Complete o pseudo-código das threads `T0`, `T1` e `T2` abaixo com a aplicação do *Padrão Mutex* conforme necessário e, ao mesmo tempo, visando explorar todo o potencial de paralelismo. A thread `T0` é a primeira a ser executada. Sua função é definir as variáveis compartilhadas por `T1` e `T2` e, em seguida, criar essas duas threads para executarem em paralelo. Todos os semáforos devem ser corretamente iniciados. Comandos na mesma linha do pseudo-código devem ser separados um por ponto-e-vírgula.

```
Thread T0:
   Variáveis compartilhadas:
      Inteiro x = 1
      Inteiro y = 1


   Criar Thread T1(x,y            )

   Criar Thread T2(x,y            )

   Iniciar T1
   Iniciar T2
```

```
Thread T1(x,y                  ):
   Variáveis locais:
      Inteiro z
   Repita para sempre:

         x = x + 1

         Aguarde entre 30 e 60 segundos

         y = y * 2

         Aguarde entre 30 e 60 segundos

         z = x - y

         Imprima z
```

```
Thread T2(x,y                  ):
   Variáveis locais:
      Inteiro z
   Repita para sempre:

         y = y + 2

         Aguarde entre 30 e 60 segundos

         x = x * 3

         Aguarde entre 30 e 60 segundos

         z = x - y

         Imprima z
```

Questão 2 .....

Considere os procedimentos `deposite` e `retire` definidos pelo pseudocódigo abaixo.

```
procedimento deposite( conta_corrente, valor ) : Lógico
   se ( valor > 0 )
      então
         saldo_conta_corrente = saldo_conta_corrente + valor
         resultado = VERDADEIRO
      senão
         resultado = FALSO
   retorne resultado

procedimento retire( conta_corrente, valor ) : Lógico
   se ( valor > 0 e valor <= conta_corrente.saldo )
      então
         conta_corrente.saldo = conta_corrente.saldo - valor
         resultado = VERDADEIRO
      senão
         resultado = FALSO
   retorne resultado
```

A thread `Caixa`, definida pelo pseudocódigo abaixo, realiza uma série de operações em contas correntes de um banco, chamando os procedimentos `deposite` e `retire`.

```
Thread Caixa
   Variáveis compartilhadas:
      CadastroContasCorrentes: lista de todas as contas correntes do banco
   repita para sempre:
      Leia( numero_conta_corrente, operacao, valor )
      conta_corrente = CadastroContasCorrentes.RecupereContaCorrente( numero_conta_corrente )
      se ( operacao = "DEPOSITO" )
         então deposite( conta_corrente, valor )
         senão retire( conta_corrente, valor )
```

Como está definida a thread `Caixa`, se duas ou mais instâncias da thread executarem em paralelo, é possível haver condição de corrida (acesso simultâneo por threads distintas a um dado compartilhado) sobre uma conta corrente, podendo gerar inconsistência no respectivo saldo. Modifique a thread com o uso de semáforos segundo o *Padrão Mutex* tal que não mais ocorram condições de corrida, independentemente do número de instâncias dessa thread que executarem em paralelo. A modificação deve maximizar o potencial de paralelismo entre as threads e é preciso deixar evidente o valor inicial de cada semáforo.

Thread `Caixa` modificada:

```
Thread Caixa
   Variáveis compartilhadas:
      CadastroContasCorrentes: lista de todas as contas correntes do banco


   repita para sempre:
      Leia( numero_conta_corrente, operacao, valor )
      conta_corrente = CadastroContasCorrentes.RecupereContaCorrente( numero_conta_corrente )
```

Questão 3 .....

Considere uma aplicação que, dada uma coleção de n listas de candidatos que prestaram o vestibular, ordena (de acordo com o número de pontos obtido por cada candidato nas provas) e imprime cada uma delas. A aplicação pode ser definida por duas threads, denominadas *Ordenação* e *Impressão*, para fins de divisão de responsabilidades e, principalmente, terminar a execução mais rapidamente, conforme mostram os algoritmos 1 e 2. Modifique esses algoritmos usando-se apenas semáforos de acordo com o *Padrão Sinalização* de forma a garantir que uma lista somente será impressa depois de devidamente ordenada.

Algoritmo 1 Ordenação

Variáveis compartilhadas:
lista: conjunto de listas de candidatos
n: número de elementos em *lista*

```
1: para i ← 1 até n faça
2:     Ordene(lista[i])
3: fim para
```

Algoritmo 2 Impressão

Variáveis compartilhadas:
lista: conjunto de listas de candidatos
n: número de elementos em *lista*

```
1: para i ← 1 até n faça
2:     Imprima(lista[i])
3: fim para
```

Questão 4 .....

Escreva o pseudocódigo de uma aplicação para fazer a soma de um conjunto de N números composta por três threads que executam paralelamente, denominadas T_1, T_2 e T_3, da seguinte forma: T_1 e T_2, cada uma, faz a soma de $N/2$ números do conjunto, enquanto T_3 recebe as somas parciais geradas por T_1 e T_2 e faz a soma total. A sincronização entre as threads deve ser feita com o uso de semáforos de acordo com o *Padrão Sinalização*.

Questão 5 .....

O *Padrão Rendezvous* segue a forma geral mostrada abaixo, na qual duas threads `A` e `B` definem um *ponto de encontro* depois de executarem o primeiro comando (`a1` e `b1`, respectivamente) e antes de executarem o segundo (`a2` e `b2`, respectivamente).

```
Thread A:
   Variáveis compartilhadas:
      aPronto, bPronto: semáforos iniciados com o valor 0
   comando a1
   aPronto.sinalizar( )
   bPronto.esperar( )
   comando a2

Thread B:
   Variáveis compartilhadas:
      aPronto, bPronto: semáforos iniciados com o valor 0
   comando b1
   bPronto.sinalizar( )
   aPronto.esperar( )
   comando b2
```

Aplique esse padrão para sincronizar três threads denominadas T_0, T_1 e T_2 definidas conforme o pseudocódigo abaixo, tal que os valores de `dx`, `dy` e `dz` sejam calculados corretamente. O padrão deve ser aplicado somente quando necessário, isto é, para pares de threads que realmente possuam uma relação de dependência para continuidade da execução.

```
Thread T_0:
   Variáveis compartilhadas:
      x: latitude; y: longitude; z: altura

   x = CalculeLatitude()



   dx = x / y

Thread T_1:
   Variáveis compartilhadas:
      x: latitude; y: longitude; z: altura

   y = CalculeLongitude()



   dy = y / z

Thread T_2:
   Variáveis compartilhadas:
      x: latitude; y: longitude; z: altura

   z = CalculeAltura()



   dz = z / x
```

Questão 6 .....

Complete o pseudo-código das threads `T0`, `T1` e `T2` abaixo com a aplicação do *Padrão Rendezvous* conforme necessário e, ao mesmo tempo, visando explorar todo o potencial de paralelismo. O valor impresso por `T1` deve ser 3, enquanto o valor impresso por `T2` deve ser 70. A thread `T0` é a primeira a ser executada. Sua função é definir as variáveis compartilhadas por `T1` e `T2` e, em seguida, criar essas duas threads para executarem em paralelo. Todos os semáforos devem ser corretamente iniciados. Comandos na mesma linha do pseudo-código devem ser separados um por ponto-e-vírgula.

```
Thread T0:
   Variáveis compartilhadas:
      Inteiro x = 1
      Inteiro y = 1
      Semaforo mutex = 1


   Criar e iniciar Thread T1(x,y,mutex           )

   Criar e iniciar Thread T2(x,y,mutex           )
```

```
Thread T1(x,y,mutex                 ):
   Variáveis locais:
      Inteiro z


   x = 10

   mutex.esperar()

   z = x - y

   Imprima z

   mutex.sinalizar()
```

```
Thread T2(x,y,mutex                 ):
   Variáveis locais:
      Inteiro z


   y = 7

   mutex.esperar()

   z = x * y

   Imprima z

   mutex.sinalizar()
```

Questão 7 .....

Complete o pseudo-código das threads `T0` e `T1` abaixo com a aplicação do *Padrão Barreira* conforme necessário e, ao mesmo tempo, visando explorar todo o potencial de paralelismo. Os valores impressos devem ser -224, -16 e 16, respectivamente por cada thread, na ordem de criação. A thread `T0` é a primeira a ser executada. Sua função é definir as variáveis compartilhadas por `T1` e, em seguida, criar `n` intâncias dessa thread para executarem em paralelo. Todos os semáforos e outras variáveis definidas devem ser corretamente iniciadas. Comandos na mesma linha do pseudo-código devem ser separados um por ponto-e-vírgula.

```
Thread T0:
   Variáveis compartilhadas:
      Inteiro x = 1
      Inteiro y = 3
      Inteiro z = 5
      Inteiro n = 3 // quantidade de threads a serem sincronizadas na barreira



   Criar e iniciar Thread T1(n,x,y,z                        )
   Criar e iniciar Thread T1(n,y,z,x                        )
   Criar e iniciar Thread T1(n,z,x,y                        )
```

```
Thread T1(n,a,b,c                      ):
   Variáveis locais:
      Inteiro z

   a = a * a
```

```
   z = a - b * c

   Imprima z
```

Questão 8 .....

A área de uma floresta tem a sua temperatura monitorada periodicamente, lendo-se a medição corrente de 100 sensores de temperatura distribuídos uniformemente nessa área. Além dos 100 valores coletados, também é feita uma normalização desses valores, divindo-se cada valor pelo maior entre eles. A leitura e a normalização em cada um dos 100 pontos de medição é realizado por uma thread específica, denominada `ObterTemperaturaNormalizada( i )`, onde i é um valor de 1 a 100 que identifica unicamente cada sensor, conforme o pseudocódigo abaixo. Ou seja, são criadas 100 threads paralelas para fazer o monitoramento da temperatura, uma para cada sensor.

```
Thread ObterTemperaturaNormalizada( i )
   Variáveis compartilhadas:
      Temperatura: lista de 100 valores reais maiores que zero
      TemperaturaNormalizada: lista de 100 valores reais maiores que zero
   repita para sempre:
      Temperatura[ i ] = LerSensor( i )
      MaiorTemperaturaLida = max( Temperatura )
      TemperaturaNormalizada[ i ] = Temperatura[ i ] / MaiorTemperaturaLida
      Salve( Temperatura )
      Salve( TemperaturaNormalizada )
      Aguarde 1 minuto
```

Como está, a thread pode não salvar corretamente os 100 valores lidos e os 100 valores calculados num mesmo ciclo de medição, bem como pode não calcular corretamente a temperatura normalizada. Aplique o padrão de sincronização por *barreira de duas fases* à thread para garantir o seu correto funcionamento.

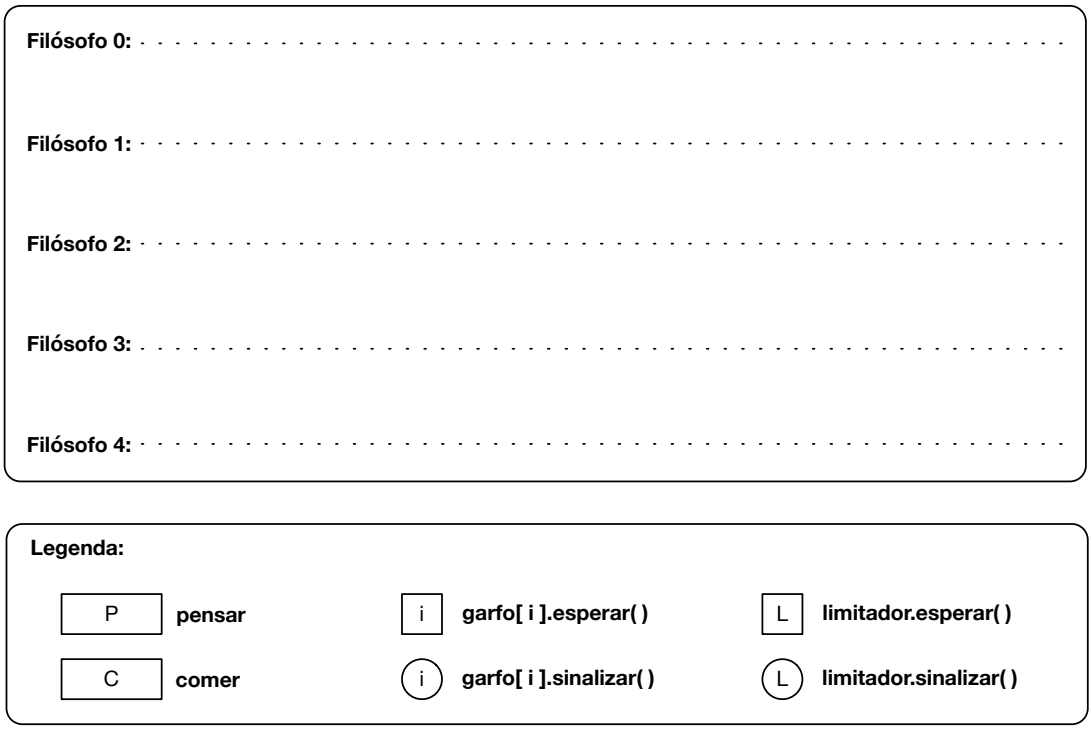
Questão 9

Construa um **caminho de execução** válido para o *Problema do Jantar dos Filósofos*, de acordo com o algoritmo 3, correspondente a cada filósofo i. Deve-se considerar um grupo de cinco filósofos (isto é, $N = 5$). O caminho de execução deve ilustrar um cenário em que os cinco filósofos comem ao menos uma vez, sendo que deve ocorrer uma situação em que três filósofos aguardam simultaneamente, enquanto os outros dois estão comendo. O caminho de execução deve ser representado graficamente, usando-se a convenção indicada na legenda do espaço para resposta (o índice i assume os valores 0, 1, 2, 3 e 4).

Algoritmo 3 Filósofo i

Variáveis compartilhadas:
N: número de filósofos
garfo: vetor de N semáforos iniciados com o valor 1, indexado de 0 a N-1
limitador: semáforo iniciado com o valor N-1

```
1:  loop
2:      pense
3:      limitador.esperar()
4:      garfo[i].esperar()              ▷ pegue o garfo à direita
5:      garfo[(i+1)%N].esperar()        ▷ pegue o garfo à esquerda
6:      coma
7:      garfo[i].sinalizar()            ▷ devolva o garfo à direita
8:      garfo[(i+1)%N].sinalizar()      ▷ devolva o garfo à esquerda
9:      limitador.sinalizar()
10: fim loop
```



Questão 10

O *Problema do Jantar dos Selvagens* é definido da seguinte forma:

Uma tribo de selvagens alimenta-se de um grande pote que pode conter até M porções. Quando um selvagem deseja comer, vai até o pote e se serve de uma porção. Caso o pote esteja vazio, o selvagem acorda o cozinheiro e espera até que o pote seja totalmente preenchido pelo cozinheiro. Depois que preenche o pote, o cozinheiro volta a dormir. Inicialmente, o pote está vazio.

Construa um **caminho de execução** válido para o *Problema do Jantar dos Selvagens*, de acordo com o pseudo-código abaixo, correspondente às threads Cozinheiro e Selvagem. Deve-se considerar um grupo de quatro selvagens e um cozinheiro. Também deve-se considerar que o pote (ou caldeirão) de comida tem capacidade para apenas duas porções, isto é, $M = 2$. O caminho de execução deve ilustrar um cenário em que os quatro selvagens servem-se ao menos uma vez e uma situação em que três selvagens aguardam simultaneamente enquanto o cozinheiro enche o pote de comida. O caminho de execução deve ser representado graficamente, usando-se a convenção indicada na legenda do espaço para resposta.

```
Variáveis compartilhadas:
   Inteiro porçõesDisponíveis = 0 // número de porções disponíveis no pote
   Semaforo mutex = 1             // acesso exclusivo ao pote
   Semaforo poteVazio = 0         // indicador de pote vazio
   Semaforo poteCheio = 0         // indicador de pote cheio
```

```
Thread Cozinheiro:
   poteVazio.esperar()
   preencherPote(M)
   poteCheio.sinalizar()
```

```
Thread Selvagem:
   mutex.esperar()
      se porçõesDisponíveis == 0
         poteVazio.sinalizar()
         poteCheio.esperar()
         porçõesDisponíveis = M
      porçõesDisponíveis = porçõesDisponíveis - 1
      servirSeDoPote()
   mutex.sinalizar()
   comer()
```

Cozinheiro: . . . . . .

Selvagem 1: . . . . . .

Selvagem 2: . . . . . .

Selvagem 3: . . . . . .

Selvagem 4: . . . . . .

Legenda:

P **preencherPote(M)**

S **servirSeDoPote( )**

v **poteVazio.esperar( )**

(v) **poteVazio.sinalizar( )**

c **poteCheio.esperar( )**

(c) **poteCheio.sinalizar( )**

m **mutex.esperar( )**

(m) **mutex.sinalizar( )**

Questão 11

Considere o **Problema da Barbearia de Hilzer** e correspondente solução, conforme descrito por Allen B. Downey, em *The Little Book of Semaphores*, p.133–136.

Construa um **caminho de execução** válido para essa solução. O caminho de execução deve ser representado graficamente usando-se a mesma convenção utilizada na ilustração de um caminho de execução para o *Problema da Barbearia*. Deve-se considerar o seguinte cenário:

- A barbearia possui 2 cadeiras para corte de cabelo.
- A barbearia possui 2 barbeiros, uma para cada cadeira.
- O limite total de clientes na barbearia é 5.

O caminho de execução deve ilustrar um cenário em que um total de 8 clientes chegam na barbearia sendo que o quarto cliente chega somente após os três primeiros serem atendidos mas, logo após o quarto cliente começar a ser atendido, todos os demais clientes (do quinto ao oitavo cliente) chegam, provocando uma fila de espera.

Questão 12 .....

Considere uma estrada de mão dupla que, em certo ponto, tem uma ponte estreita, de forma que não caibam dois carros lado a lado. Carros que estejam indo no mesmo sentido (Norte ou Sul) podem usar a ponte ao mesmo tempo, mas carros que estejam em sentidos opostos não, ou seja, uma carro deve aguardar para entrar na ponte enquanto essa estiver ocupada com carros vindo em sentido oposto. Nas extremidades Sul e Norte da ponte, há controladores, respectivamente denominados `ControladorSul` e `ControladorNorte`. Os controladores revezam no tempo para autorizar carros entrarem na ponte. Quando um controlador autoriza o fluxo, deve deixar todos os carros que estejam aguardando passar, mas deve limitar esse número a 10 em cada vez; caso chegue a esse número ou terminem os carros em espera, deve interromper o fluxo e passar a vez para o outro controlador. Além disso, cada controlador deve garantir que, a cada momento, o número máximo de carros passando sobre a ponte é 5 (cinco). Supondo que cada carro (vindo pelo Sul ou pelo Norte) e que cada controlador sejam processos, defina o pseudo-código dos processos `ContradorSul` e `CarroSul` (um carro vindo pelo Sul) que implementem essa semântica com o uso de *semáforos*. Pode-se assumir que assim que um carro termina de atravessar a ponte o mesmo sinaliza o controlador que o autorizou.