Trabalho Prático de Computação Concorrente (PESO 2,0 NA NOTA FINAL)

ATENÇÃO: O TRABALHO OBRIGATORIAMENTE TEM QUE SER FEITO EM GRUPOS DE **3 (três) INTEGRANTES**. TRABALHOS COM MENOS OU MAIS DO QUE 3 ALUNOS **NÃO SERÃO CORRIGIDOS** NEM CONSIDERADOS NA NOTA (EXCEÇÕES DEVEM SER TRATADAS COM A PROFESSORA ANTES DA DATA DA ENTREGA)

LINGUAGEM DE IMPLEMENTAÇÃO DO TRABALHO: JAVA OU C (Código contendo documentação detalhada de todas as variáveis e funções/métodos)

AULAS LIVRES PARA OS GRUPOS FAZEREM O TRABALHO: 10/10, 17/10, 9/11.

DATA DA ENTREGA: **16/11** (INADIÁVEL, ATÉ A MEIA-NOITE VIA GOOGLE DRIVE – Pasta "Trabalho Pratico")

1. Descrição do problema

Um **sistema de reserva de assentos** consiste em um programa que mantém informações sobre a alocação de assentos em um espaço físico ou em um meio de transporte, permitindo que usuários remotos possam ver o estado das alocações, solicitar nova alocação ou cancelar uma alocação realizada previamente pelo próprio usuário. Normalmente esse tipo de sistema é executado em um ambiente distribuído com duas aplicações distintas: uma que executa o código que gerencia as estruturas de dados usadas para manter o estado de alocação dos assentos e outra que é executada em máquinas remotas pelos usuários. Em essência, trata-se de uma aplicação concorrente com vários fluxos de execução independentes.

1.1. Programa principal

Neste trabalho, vamos projetar e implementar uma versão simplificada desse tipo de sistema o qual executará em uma única máquina com várias threads independentes. Os assentos deverão ser representados por estruturas de dados convencionais, como vetores unidimensionais ou bidimensionais. O número de assentos deve ser passado como parâmetro no início da execução da aplicação. Os estados possíveis de um assento são: **livre** ou **reservado**. Quando um assento é reservado, ele deve guardar um identificador único do usuário que fez a reserva no mapa de assentos.

O usuário remoto poderá executar as seguintes operações:

1. Visualizar o estado corrente de alocações dos assentos;
2. Selecionar um assento livre para alocação;
3. Cancelar uma reserva feita por ele anteriormente.

Dois ou mais usuários não podem alocar o mesmo assento ao mesmo tempo. Se um assento já está alocado, ele não pode ser realocado para outro usuário, apenas se esse assento for liberado posteriormente pelo usuário dono da reserva. Dois ou mais usuários podem visualizar o estado das alocações ao mesmo tempo, mas apenas um usuário de cada vez pode alterar o estado de um determinado assento. Entre o instante em que um usuário visualiza o mapa de assentos e o instante seguinte em que ele solicita a alocação de um assento, pode ocorrer de outro usuário já ter alocado o assento que previamente (na visualização anterior) estava livre.

1.2. Arquivo de log

Uma parte importante deste trabalho será a geração de um log de saída para registrar todas as operações realizadas no sistema e a implementação de um programa auxiliar para verificar se a execução ocorreu com sucesso.

Para isso, as threads deverão registrar cada ação realizada, juntamente com o mapa de assentos corrente, antes que outra operação altere esse mapa. Esse registro deverá ser feito em um **buffer de log da aplicação** usando o padrão **produtor/consumidor**. As threads que representam as ações dos usuários serão `produtores`, registrando cada operação no buffer. Uma thread adicional, **com identificador 0**, será a thread `consumidora` cuja tarefa consiste em escrever o buffer de log em um arquivo de saída, preservando a ordem de inserção dos registros no buffer de log. Cada elemento do buffer deverá ser escrito em uma linha separada do arquivo de saida.

A thread principal do programa deverá inicializar as estruturas de dados e variáveis necessárias e disparar as demais threads que representarão os usuários remotos e a thread de escrita do log. Quando todas as threads de usuários remotos terminarem suas execuções e o log for completamente escrito no arquivo de saída, o programa deverá ser encerrado.

Os códigos executados pelas threads (principal, de log e usuárias) poderão ser implementados na linguagem C (usando a biblioteca Pthreads) ou em Java.

1.3. Tipos e funções que deverão ser implementadas

Os seguintes tipos de dados deverão ser definidos:

- `t_Assentos`: estrutura de dados para representar todos os assentos gerenciados (pode ser um vetor unidimensional ou bidimensional);
- `t_Assento`: estrutura de dados para representar unicamente um assento.

As threads (usuários remotos) deverão ser identificadas unicamente usando um **número inteiro** a partir do valor **1**. A instância da estrutura de dados que representa os assentos deverá ser compartilhada entre todas as threads usuárias.

Todos os assentos devem ser inicializados com estado "livre" (valor 0).

As seguintes funções deverão ser implementadas:

1. `visualizaAssentos()`: exibe o mapa de assentos com seus estados. Armazena no buffer de log o código da operação (1), o identificador da thread que a realizou e o mapa de assentos com seus estados lidos.

2. `int alocaAssentoLivre(t_Assento * assento, int id)`: escolhe, de forma randômica, um assento livre e o aloca para o usuário com identificador `id`. Apenas a thread de mesmo identificador `id` pode realizar essa alocação. O assento alocado é retornado no primeiro parâmetro (variável `assento` passada por referência). A função retorna `1` se um assento foi alocado e `0` caso contrário (apenas quando todos os assentos já estavam ocupados). Armazena no buffer de log o código da operação (2), o identificador da thread que a realizou, o assento selecionado e o mapa de assentos imediatamente após a alocação. Se todos os assentos já estavam alocados, o novo mapa deverá permaner igual ao anterior.
3. `int alocaAssentoDado(t_Assento assento, int id)`: tenta alocar o assento dado (`assento`) para o usuário com identificador `id`. Apenas a thread de mesmo identificador `id` pode realizar essa alocação. A função retorna `1` se o assento foi alocado e `0` caso contrário (apenas quando o assento já estava ocupado). Armazena no buffer de log o código da operação (3), o identificador da thread que a realizou, o assento selecionado e o mapa de assentos imediatamente após a alocação. Se o assento escolhido já estava alocado, o novo mapa deverá permaner igual ao anterior.

4. `liberaAssento(t_Assento assento, int id)`: libera/desaloca o assento (`assento`) dado alocado pelo usuário de identificador `id`. Apenas a thread com esse mesmo identificador pode liberar o assento. A função retorna `1` se o assento foi desalocado com sucesso e `0` caso contrário (quando o assento estava livre ou ocupado por outro usuário). Armazena no buffer de log o código da operação (4), o identificador da thread que a realizou, o assento selecionado e o mapa de assentos imediatamente após a alocação. Se o assento escolhido não estava alocado ou já estava alocado mas por outro usuário, o novo mapa deverá permaner igual ao anterior.

1.4. Código das threads

Diferentes versões do código das threads usuárias **deverão** ser implementadas alterando-se a ordem e o número de operações solicitadas. Os pseudocódigos abaixo ilustram essa ideia definindo três implementações distintas para o código das threads.

```
void * thread1 (void * arg) {
   //salva seu identificador unico
   int tid = * (int*) arg;
   t_Assento assento;
   //visualiza mapa de assentos
   visualizaAssentos();
   //tenta alocar um assento livre
   alocaAssentoLivre(&assento, tid);
   //visualiza mapa de assentos
   visualizaAssentos();
   pthread_exit(NULL);
}

void * thread2 (void * arg) {
   //salva seu identificador unico
   int tid = * (int*) arg;
   //visualiza mapa de assentos
   visualizaAssentos();
   t_Assento assento = ...//inicializa com um assento
   //tenta alocar um assento especifico
   alocaAssentoDado(assento, tid);
   //visualiza mapa de assentos
   visualizaAssentos();
   pthread_exit(NULL);
}

void * thread3 (void * arg) {
   //salva seu identificador unico
   int tid = * (int*) arg;
   t_Assento assento;
   //visualiza mapa de assentos
   visualizaAssentos();
   //tenta alocar um assento livre



   alocaAssentoLivre(&assento, tid);
   //visualiza mapa de assentos
   visualizaAssentos();
   //libera alocacao
   liberaAssento(assento, tid);
   //visualiza mapa de assentos
   visualizaAssentos();
   pthread_exit(NULL);
}
```

2. Entrada

Para a execução do programa, os seguintes dados deverão ser fornecidos como entrada, nesta ordem:

```
<nome do arquivo de log de saída> <quantidade de assentos>
```

3. Exemplo de arquivo de saída

Considerando uma execução do programa que dispara as três threads mostradas na seção "Código das threads" uma única vez, podemos ter o seguinte arquivo de saída (considerando que a thread 1 executa completamente, depois a thread 2 e finalmente a thread 3 e que o número de assentos é 5):

```
1,1,[0,0,0,0,0]
2,1,1,[1,0,0,0,0]
1,1,[1,0,0,0,0]
1,2,[1,0,0,0,0]
3,2,4,[1,0,0,2,0]
1,2,[1,0,0,2,0]
1,3,[1,0,0,2,0]
2,3,2,[1,3,0,2,0]
1,3,[1,3,0,2,0]
4,3,2[1,0,0,2,0]
```

O programa auxiliar que irá avaliar o log de saída e validar a execução do programa principal deverá reproduzir, de forma sequencial, a execução de cada função registrada no log e avaliar se as operações foram executadas corretamente garantindo a correta transição de estados do mapa de assentos.

No formato de saída mostrado acima, o programa auxiliar deverá ler cada linha do arquivo de saída e realizar o processamento correspondente ao conteúdo da linha. A linguagem usada para implementar o programa auxiliar é de livre escolha. (Pode ser C, Java, ou linguagens de scripting como Lua, Python, etc..)

Escolhida a linguagem de implementação do programa auxiliar, uma outra alternativa para gerar o arquivo de log é escrevê-lo usando a sintaxe da própria linguagem e depois apenas executá-lo.

Como exemplo, considere que escolhemos a linguagem C para implementar o programa auxiliar. Nesse caso o arquivo de log acima poderia ser reescrito dessa forma:

```
int main() {
   int v1[] = {0,0,0,0,0}; op1(1,v1);
   int v2[] = {1,0,0,0,0}; op2(1,1,v2);
   int v3[] = {1,0,0,0,0}; op1(1,v3);
   int v4[] = {1,0,0,0,0}; op1(2,v4);
   int v5[] = {1,0,0,2,0}; op3(2,4,v5);
   int v6[] = {1,0,0,2,0}; op1(2,v6);
   int v7[] = {1,0,0,2,0}; op1(3,v7);
   int v8[] = {1,3,0,2,0}; op2(3,2,v8);
   int v9[] = {1,3,0,2,0}; op1(3,v9);
   int v10[] = {1,0,0,2,0}; op4(3,2,v10);
   return 0;
}
```

assumindo-se que em outro arquivo .c definimos as funções chamadas (uma para cada operação permitida):

```
#include<stdio.h>
int buffer[] = {0,0,0,0,0};
void op1(int tid, int vet[]) { //operação 1
  //compara se o vetor passado é igual ao buffer corrente
}
void op2(int tid, int pos, int vet[]) { //operação 2
  //verifica se a operação é válida, altera o buffer
  //corrente de acordo com a operação solicitada e compara
  //o vetor passado com o buffer corrente
}
void op3(int tid, int pos, int vet[]) { //operação 3
  //...
}
void op4(int tid, int pos, int vet[]) { //operação 4
  //...
}
```

Nesse novo formato, o arquivo de log pode ser executado diretamente, dado que as funções chamadas por ele estão implementadas em outro arquivo.

A escolha entre essas duas alternativas colocadas é livre. O programa auxiliar deve gerar como saída uma mensagem que valida a execução do programa principal ou uma mensagem de erro apropriada indicando em qual operação ocorreu erro de.

4. **Etapas do trabalho**

A execução do trabalho deverá ser organizada nas seguintes etapas:

1. Compreender o problema principal, pensar e esboção uma solução concorrente para o msmo
2. Projetar as estruturas de dados e as funções/métodos que deverão ser implementados
3. Projetar e implementar o programa auxiliar que deverá avaliar o log de execução do programa principal
4. Construir um conjunto de casos de teste para avaliação da aplicação
5. Implementar a solução projetada, avaliar a corretude dos programas, refinar a implementação e refazer os testes
6. Redigir os relatórios e incluir documentação adicional nos códigos (se necessário)

5. Artefatos que deverão ser entregues

1. Relatório: documentação do projeto da solução (esboço das estruturas de dados e lógica principal dos algoritmos implementados), testes realizados e resultados obtidos. O relatório deverá conter informações suficientes para o professor compreender a solução proposta sem precisar recorrer ao código fonte do programa.
2. Código fonte e roteiro para compilação e execução dos programas.

6. **Data de entrega e critérios de avaliação**

O trabalho deverá ser feito em TRIPLA e entregue até o dia **16 de novembro** de 2017. Até o dia 10 **de OUTUBRO** deve ser enviado email com as composições dos grupo. No dia **17** ou **19 de OUTUBRO** os grupos deverão mostrar para a professora em sala de aula os seus Projetos da solução (item 2 nas etapas acima). Os seguintes itens serão avaliados com o respectivo peso:

• Compilação e execução correta dos dois programas: 5 pontos

• Relatório (incluindo projeto da solução e testes realizados): 3 pontos

• Modularidade, organização e documentação do código fonte, estratégias de concorrência: 2 pontos

Não é permitido copiar a solução do trabalho de outros colegas ou de terceiros. Os alunos integrantes da equipe poderão ser chamados pelo professor para apresentar/explicar o trabalho a qualquer aula antes do prazo de entrega.