Trabalho – Valor: 10%

- O prazo de entrega será 01/07/2021 até às 23:59 horas, com entrega via Moodle.

Os objetivos deste trabalho são:

- Sobre *threads*:
  - Entender como manipular *threads usando* a biblioteca `pthread.h`;
  - Entender como fazer *Inter Process Communication* (IPC) entre as *threads* usando 1) mutex e 2) semáforo;
  - Entender como sincronizar as *threads* de forma que se evitem condições de corrida, *deadlock* ou *starvation*;
  - Entender a analogia do *buffer* como uma caixa d'água furada com M produtores e N consumidores;
  - Entender como se aplicam *mutexes* (*Mutual Exclusions*) e semáforos usando `pthread_mutex_t` e `sem_t` presentes nas bibliotecas `pthread.h` e `semaphore.h`;
  - Observar o *overhead* na criação e no escalonamento de *threads* em tarefas paralelas.

Para atender aos objetivos, a dupla de alunos deve implementar em linguagem C (só para reforçar, não é C++) dois programas separados que abordam dois problemas diferentes de IPC.

1. Problema de M produtores e N consumidores:
   a. O usuário deverá inserir em *runtime* as quantidades de produtores (M) e consumidores (N), além do tamanho do *buffer*:
      i. M e N obviamente devem ser valores positivos, assim como o tamanho do *buffer*;
      ii. O *buffer* é um vetor do tipo `int`, alocado usando `malloc()` e deve ser preenchido com `-1`.
   b. Os produtores e os consumidores deverão inserir/retirar elementos do *buffer* indefinidamente. O programa só terá a sua execução finalizada de maneira involuntária via CTRL-C. As regras de produção/consumo são:
      i. Os itens serão produzidos/consumidos usando a ordenação FILO (*first-in, last-out*), como em uma pilha. Em outras palavras, deve-se armazenar qual foi o último *slot* de *buffer* utilizado;
      ii. O processo de produção é simplesmente inserir um valor inteiro aleatório positivo e incrementar o contador de *slots* utilizados. Porém, utilize `sleep(rand() % 5)` para simular uma produção que demora entre zero e dois segundos;
      iii. Como não há destinação para os itens consumíveis, o processo de consumo também será simulado. Para tanto, o item consumido assumirá o valor `-1` e o contador de *slots* utilizados deve ser decrementado. Utilize `sleep(rand() % 2)`, para simular um consumo que provavelmente será mais rápido do que a produção;
      iv. Após a produção/consumo do item, o *buffer* deverá ser impresso na tela e disposto na vertical, com um elemento por linha. Considere `-1` como vazio e essas posições do buffer devem ser impressas apenas como um espaço em branco.
   c. Deve-se, obrigatoriamente, utilizar semáforos para sinalizar produtores e consumidores que podem realizar a sua tarefa. Portanto, os M produtores e os N consumidores deverão ser executados assincronamente, onde cada um deles possui a sua própria *thread*.
   d. Dicas:
      i. Implemente como variáveis globais:
         1. Um semáforo para sinalizar os produtores;
         2. Um semáforo para sinalizar os consumidores;
         3. Um *mutex* para o contador de slots ocupados;
         4. A variável que armazena o tamanho do *buffer*;

5. A variável que armazena o número de *slots* ocupados (em outras palavras, indica o próximo *slot* livre).

e. Saída de exemplo:

```
##Produtor##
Próximo slot livre:1
[0]     1957747793
[1]     .
[2]     .
[3]     .
[4]     .
[5]     .
[6]     .
[7]     .
[8]     .
[9]     .
Item produzido: 1957747793

##Produtor##
Próximo slot livre:2
[0]     1957747793
[1]     1649760492
[2]     .
[3]     .
[4]     .
[5]     .
[6]     .
[7]     .
[8]     .
[9]     .
Item produzido: 1649760492
```

```
##Produtor##
Próximo slot livre:3
[0]     1957747793
[1]     1649760492
[2]     1025202362
[3]     .
[4]     .
[5]     .
[6]     .
[7]     .
[8]     .
[9]     .
Item produzido: 1025202362

##Consumidor##
Próximo slot livre: 2
[0]     1957747793
[1]     1649760492
[2]     .
[3]     .
[4]     .
[5]     .
[6]     .
[7]     .
[8]     .
[9]     .
Item consumido: 1025202362
```

```
##Produtor##
Próximo slot livre:3
[0]     1957747793
[1]     1649760492
[2]     783368690
[3]     .
[4]     .
[5]     .
[6]     .
[7]     .
[8]     .
[9]     .
Item produzido: 783368690

##Consumidor##
Próximo slot livre: 2
[0]     1957747793
[1]     1649760492
[2]     .
[3]     .
[4]     .
[5]     .
[6]     .
[7]     .
[8]     .
[9]     .
Item consumido: 783368690
```

2. Problema *multithread* de soma de vetores:
   a. Considere três vetores, `x`, `y` e `z`, todos de tamanho fixo: 160 milhões de elementos (16×10^7) em cada vetor;
   b. Os vetores são do tipo *float* e devem ser alocados usando `malloc()` antes de se iniciar o processo;
   c. O número de threads (N) deve ser definido em *runtime* pelo usuário. A única restrição é a de que 16×10^7 deve ser um múltiplo de N, pois os vetores serão divididos em N partes, uma para cada *thread*;
   d. Como o processamento paralelo em N partes, deve-se criar N *mutexes* para controlar a entrada na região crítica no seu código.
   e. Deve haver a *thread* `void* preencheVetores(void* argPtr)` que será responsável por preencher os vetores `x` e `y` com valores aleatórios entre zero e um, cujos parâmetros de entrada são:
      i. `float* vetor` – ponteiro para o início do vetor que será preenchido;
      ii. `unsigned int posInicial` – posição inicial de preenchimento do vetor;
      iii. `unsigned int posFinal` – posição final de preenchimento do vetor;
      iv. `unsigned int contMutexThread` – contador que indicará qual dos N *mutexes* deve ser utilizado para controlar a região crítica da *thread*;
   f. Deve haver a *thread* `void* somaVetores(void* argPtr)` que será responsável por produzir o vetor `z` a partir da soma `x` e `y`. Os parâmetros de entrada são:
      i. `float* x` – ponteiro para o início do vetor `x`;
      ii. `float* y` – ponteiro para o início do vetor `y`;
      iii. `float* z` – ponteiro para o início do vetor `z`;
      iv. `unsigned int posInicial` – posição inicial de preenchimento do vetor;

v. `unsigned int posFinal` – posição final de preenchimento do vetor;

vi. `unsigned int contMutexThread` – contador que indicará qual dos *N mutexes* deve ser utilizado para controlar a região crítica da *thread*;

g. O código deve mostrar na tela quais as posições dos vetores que serão processadas por cada *thread*.

h. Deve-se cronometrar o tempo de execução. Para isso, utilize um timer do tipo `time_t` (que tem um segundo de resolução) e é implementado em `time.h`. O tempo de cronometragem deve ser entre antes de se iniciar o processamento e depois de se finalizarem todas as *threads*.

i. Dicas:

i. Implemente os *mutexes* como um vetor de *N* posições a ser alocado via `malloc()`;

ii. Os *mutexes* os vetores `x`, `y` e `z` devem ser implementados como variáveis globais, que serão compartilhadas entre todas as threads em execução;

iii. Por limitação da biblioteca `pthread.h`, as *threads* só podem ter um argumento de entrada do tipo `void`, como vocês podem ver nos protótipos acima. Por isso, deve-se criar dois *structs* (um para cada tipo de *thread*) que servirão para passar os parâmetros.

j. Saída de exemplo:

```
Deseja utilizar quantas threads?
--> 16
Qual o tamanho dos vetores a serem somados?
--> 160000000
Qual o número de repetições (valores inteiros não-negativos) a serem feitas?
--> 1
A thread 0 processará os elementos dos vetores nas posições de 0 a 9999999
A thread 1 processará os elementos dos vetores nas posições de 10000000 a 19999999
A thread 2 processará os elementos dos vetores nas posições de 20000000 a 29999999
A thread 3 processará os elementos dos vetores nas posições de 30000000 a 39999999
A thread 4 processará os elementos dos vetores nas posições de 40000000 a 49999999
A thread 5 processará os elementos dos vetores nas posições de 50000000 a 59999999
A thread 6 processará os elementos dos vetores nas posições de 60000000 a 69999999
A thread 7 processará os elementos dos vetores nas posições de 70000000 a 79999999
A thread 8 processará os elementos dos vetores nas posições de 80000000 a 89999999
A thread 9 processará os elementos dos vetores nas posições de 90000000 a 99999999
A thread 10 processará os elementos dos vetores nas posições de 100000000 a 109999999
A thread 11 processará os elementos dos vetores nas posições de 110000000 a 119999999
A thread 12 processará os elementos dos vetores nas posições de 120000000 a 129999999
A thread 13 processará os elementos dos vetores nas posições de 130000000 a 139999999
A thread 14 processará os elementos dos vetores nas posições de 140000000 a 149999999
A thread 15 processará os elementos dos vetores nas posições de 150000000 a 159999999
Tempo decorrido: 11 segundos
```

k. Saída de exemplo (com impressão apenas para depuração):

```
Deseja utilizar quantas threads?
--> 16
Qual o tamanho dos vetores a serem somados?
--> 32
A thread 0 processará os elementos dos vetores nas posições de 0 a 1
A thread 1 processará os elementos dos vetores nas posições de 2 a 3
A thread 2 processará os elementos dos vetores nas posições de 4 a 5
A thread 3 processará os elementos dos vetores nas posições de 6 a 7
A thread 4 processará os elementos dos vetores nas posições de 8 a 9
A thread 5 processará os elementos dos vetores nas posições de 10 a 11
A thread 6 processará os elementos dos vetores nas posições de 12 a 13
A thread 7 processará os elementos dos vetores nas posições de 14 a 15
A thread 8 processará os elementos dos vetores nas posições de 16 a 17
A thread 9 processará os elementos dos vetores nas posições de 18 a 19
A thread 10 processará os elementos dos vetores nas posições de 20 a 21
```

```
A thread 11 processará os elementos dos vetores nas posições de 22 a 23
A thread 12 processará os elementos dos vetores nas posições de 24 a 25
A thread 13 processará os elementos dos vetores nas posições de 26 a 27
A thread 14 processará os elementos dos vetores nas posições de 28 a 29
A thread 15 processará os elementos dos vetores nas posições de 30 a 31
Tempo decorrido: 0 segundos
[0]     x = 0.840188    y = 0.783099    z = 1.623287
[1]     x = 0.394383    y = 0.798440    z = 1.192823
[2]     x = 0.335223    y = 0.911647    z = 1.246870
[3]     x = 0.768230    y = 0.197551    z = 0.965781
[4]     x = 0.277775    y = 0.477397    z = 0.755172
[5]     x = 0.553970    y = 0.628871    z = 1.182841
[6]     x = 0.364784    y = 0.952230    z = 1.317014
[7]     x = 0.513401    y = 0.916195    z = 1.429596
[8]     x = 0.635712    y = 0.141603    z = 0.777314
[9]     x = 0.717297    y = 0.606969    z = 1.324266
[10]    x = 0.016301    y = 0.137232    z = 0.153532
[11]    x = 0.242887    y = 0.804177    z = 1.047063
[12]    x = 0.156679    y = 0.129790    z = 0.286470
[13]    x = 0.400944    y = 0.108809    z = 0.509753
[14]    x = 0.998924    y = 0.512932    z = 1.511857
[15]    x = 0.218257    y = 0.839112    z = 1.057369
[16]    x = 0.612640    y = 0.637552    z = 1.250192
[17]    x = 0.296032    y = 0.524287    z = 0.820319
[18]    x = 0.493583    y = 0.292517    z = 0.786100
[19]    x = 0.972775    y = 0.771358    z = 1.744133
[20]    x = 0.526745    y = 0.400229    z = 0.926974
[21]    x = 0.769914    y = 0.891529    z = 1.661443
[22]    x = 0.283315    y = 0.807725    z = 1.091039
[23]    x = 0.352458    y = 0.919026    z = 1.271485
[24]    x = 0.069755    y = 0.525995    z = 0.595751
[25]    x = 0.949327    y = 0.086056    z = 1.035383
[26]    x = 0.192214    y = 0.890233    z = 1.082446
[27]    x = 0.663227    y = 0.348893    z = 1.012120
[28]    x = 0.064171    y = 0.457702    z = 0.521873
[29]    x = 0.020023    y = 0.063096    z = 0.083119
[30]    x = 0.238280    y = 0.902208    z = 1.140488
[31]    x = 0.970634    y = 0.850920    z = 1.821554
```