Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
Disciplina: Programação concorrente 2019.2 Reposição Prova 1
Professor: Thiago Emmanuel Pereira
Aluno:

obs 1) Pequenas mudanças nas APIs indicadas podem ser realizada caso permitam uma melhor implementação. Indique na prova o motivo da mudança. Tente evitar soluções que possam apresentar péssimo desempenho (p.ex spin locks indevidos).
obs 2) Embora não seja essencial, tente seguir as APIs de C e Java o mais próximo possível. Caso tenha esquecido, indique na prova a sua "versão" da API que está sendo considerada.

1. (java) A classe CyclicBarrier, discutida em sala, é uma abstração do pacote util.concurrent que permite que um conjunto de threads esperem umas pelas outras (a chamada, barreira). A barreira é dita cíclica porque pode ser reusada após a liberação das threads que estavam em espera. Implemente a API abaixo. Você pode usar o constructo **synchronized** (em métodos e blocos), Locks, variáveis condicionais (wai/notify/notifyAll) e semáforos.

   CyclicBarrier (int numThreads) *- Este construtor indicar que a instância da classe criada será baseada num conjunto de* ***numThreads*** *threads. Ou seja, a barreira será derrubada após uma quantidade de threads igual à* ***numThreads*** *ter chamado o método* ***await.***

   await() *- Uma thread bloqueia ao chamar esse método até que todas as outras threads (até um máximo de* ***numThreads****) também tenham chamado o mesmo método.*

2. (clang) Considere a API e o tipo ***shylock_t*** abaixo. Esta abstração define uma espécie de lock que permite que um número máximo de threads acesse uma região crítica, de maneira concorrente. A API define três funções, no estilo da API de locks de C: 1) uma função **init**, para construir o objeto de maneira adequada; 2) uma função **acquire**, que permite/nega o acesso à região crítica; e 3) uma função **release**, que libera um lock obtido anteriormente. O número máximo de threads que podem executar de maneira concorrente na região crítica é definida por um parâmetro na função init. Se este número máximo já tiver sido atingido, uma thread que executa a função acquire deve bloquear até que uma outra thread libere o lock. Implemente a API indicada. Você pode usar as construções básicas vistas em sala de aula (locks, variáveis condicionais e semáforos). Você pode mudar também a definição da estrutura **shylock_t**

   ```
   typedef struct _shylock_t {
        int max_threads;
   } shylock_t;

   void shylock_init (shylock_t *lock, int max_nthreads);
   void shylock_acquire (shylock_t *lock);
   void shylock_release (shylock_t *lock);
   ```

3. (clang) Abaixo, temos um esboço de implementação de um Fila. Esta implementação tem problemas de concorrência. Detecte e corrija os problemas detectados. Além de critérios de corretude, considere critérios de desempenho; ou seja, tente proteger somente trechos de código que façam parte da região crítica.

```
// basic node structure
typedef struct __node_t {
      int value;
      struct __node_t  *next;
} node_t;

typedef struct __queue_t {
      node_t     *head;
      node_t     *tail;
} list_t;

void Queue_Init(queue_t    *Q) {
      note_t *tmp = malloc(sizeof(node_t));
      tmp->next = NULL;
      q->head = q->tail = tmp;
}

void Queue_Enqueue(queue_t *q, int value) {
      node_t *tmp = malloc(sizeof(node_t));
      assert(tmp != NULL);
      tmp->value = value;
      tmp->next = NULL;
      q->tail->next = tmp;
      q->tail = tmp;
}

int Queue_Dequeue(queue_t *q, int *value) {
      node_t *tmp = q->head;
      node_t *new_head = tmp->next;
      if (new_head == NULL) {
            return -1; // queue was empty
      }
      *value = new_head->value;
      q->head = new_head;
      free(tmp);
      return 0;
}
```