Módulo 2 - Lista de Exercícios 2 (2021/1 REMOTO)

Carlos Bravo
119136241

Setembro 2021

Questão 1

(a) A parte realizada pela main já está comentada, então passando para o algoritmo concorrente.

A barreira é uma forma de bloquear todas as threads até que estejam no mesmo ponto do código, ela possui um contador que vai somando e bloqueando a thread. Quando o número de threads bloqueadas é nthreads-1 a barreira é liberada e todas as threads podem seguir, zerando o contador.

A ideia do algoritmo implementado na tarefa é fazer uma "soma logarítmica", primeiro somando o elemento anterior, depois os 3 anteriores, depois os 7 anteriores e assim até somar todos em cada posição. A variável salto irá percorrer o vetor em potências de 2. Se o id está depois do corte realizado, irá pegar o valor da posição id-salto, esperar as outras threads, modificar o valor atual somando e esperar as outras threads novamente, passando pro próximo corte.

(b) Sim, esse algoritmo funciona pois o valor na posição i depende do valor da posição $i-1$, mas ambos possuem a soma até a posição $i-2$. Análogo a como em um fatorial é possível reaproveitar valores anteriores. Dessa forma o algoritmo primeiro irá somar a posição anterior. Para obter a soma das 3 anteriores, a posição $i-1$ já foi somada no primeiro passo, e o valor que se encontra na posição $i-2$ é a soma dos valores $i-2$ e $i-3$. Assim, é possível pegar esse valor e obter a soma das 3 posições anteriores. Seguindo esse mesmo raciocínio, é possível pegar as somas nas potências de 2 anteriores a uma posição.

(c) A primeira barreira tem o objetivo de garantir que cada thread consiga pegar um valor anterior antes dele ser alterado no próximo passo. A segunda barreira tem o objetivo de garantir que cada thread consiga alterar o valor antes dele ser lido por outra thread no próximo passo. Se remover qualquer uma das duas barreiras poderá haver condições de corrida, com threads lendo valores durante uma alteração.

Questão 2

```
// Condicao de ser multiplo de 100
pthread_cond_t cond;
pthread_mutex_t mutex;
long long int contador = 0;
```

```
void *T1 (void * arg){
  while(1){
    FazAlgo(contador);
    // Trava para poder alterar a variavel global
    pthread_mutex_lock(&mutex);
    contador++;
    // Se for multiplo libera a condicao e espera
    if(contador % 100 == 0){
      pthread_cond_signal(&cond);
      pthread_cond_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex);
  }
}

void *T2(void *arg){
  while (1){
    // Trava para poder ler a variavel global sem ser alterada
    pthread_mutex_lock(&mutex);
    // Se for multiplo imprime e libera a condicao para T1
    if(contador % 100 == 0){
      printf("%lld\n",contador);
      pthread_cond_signal(&cond);
    }
    // Espera a condicao estar liberada novamente
    pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
  }
}
```

É necessário um if em T2 ao invés de começar com wait, mesmo com o if em T1, pois se a variável contador chegar a 100 e T2 não tiver começado, irá direto para o wait enquanto T1 já está esperando, entrando num deadlock.

Questão 3

(a) As tarefas a serem executadas são salvas em uma lista encadeada. As threads são criadas e ficam em loop, esperando uma tarefa para realizar. Se a thread está livre e há uma tarefa na lista, começa a executar a primeira. Quando todas as tarefas são realizadas e o pool recebe o aviso de shutdown, as threads são finalizadas.

(b) O erro ocorre quando as tarefas são finalizadas mas não há o aviso de shutdown antes de todas as threads serem pausadas. Pode ser que após a criação das tarefas, a thread da main seja pausada e as threads do pool consigam realizar todas as tarefas, entrando em modo de espera. No entanto, quando a main retornar e der o aviso de shutdown, irá esperar as threads acabarem pelo *join*, no entanto, elas estão travadas, então o código não irá terminar. Para solucionar esse problema, quando der o aviso de shutdown é possível notificar todas as threads para voltarem a funcionar, saindo da condição de espera e indo para a condição de finalização.

```
public void shutdown() {
    synchronized(queue) {
      this.shutdown=true;
      queue.notifyAll();
    }
    for (int i=0; i<nThreads; i++)
      try { threads[i].join(); }
      catch (InterruptedException e) {return;}
}
```

Questão 4

(a) A inanição pode ocorrer quando há sempre leitores lendo. Um caso em que isso ocorre é quando o tempo de leitura é muito longo ou há muitas threads leitoras. Assim, antes de um leitor terminar de ler, outros leitores começarão, nunca permitindo uma brecha para os escritores.

(b)

```
public class LE{
  private int leit; // Quantidade de leitores lendo
  private int escr; // Quantidade de escritores escrevendo
  private int escrQuer; // Quantos escritores querem escrever

  public LE(){
    this.leit = 0;
    this.escr = 0;
    this.escrQuer = 0;
  }

  // Entrada de leitores
  public synchronized void EntraLeitor (int id) {
    try {
      // Enquanto tiver um escritor querendo escrever, se bloqueia
      while (this.escrQuer > 0) {
        System.out.println ("le.leitorBloqueado("+id+")");
        wait();
      }
      this.leit++; // Leitor lendo
      System.out.println ("le.leitorLendo("+id+")");
    } catch (InterruptedException e) { }
  }

  // Saida de leitores
  public synchronized void SaiLeitor (int id) {
    this.leit--; // Saiu o leitor
    // Se nao tiver mais leitores, libera apenas um escritor
    if (this.leit == 0)
```

```
      this.notify();
    System.out.println ("le.leitorSaindo("+id+")");
  }

  // Entrada de escritores
  public synchronized void EntraEscritor (int id) {
    try {
      this.escrQuer++; // Escritor quer escrever
      // Enquanto tiver alguem lendo ou escrevendo, se bloqueia
      while ((this.leit > 0) || (this.escr > 0)) {
        System.out.println ("le.escritorBloqueado("+id+")");
        wait();
      }
      this.escr++; // Escritor escrevendo
      System.out.println ("le.escritorEscrevendo("+id+")");
    } catch (InterruptedException e) { }
  }

  // Saida de escritores
  public synchronized void SaiEscritor (int id) {
    this.escr--; // Escritor terminou
    this.escrQuer--; // Escritor nao quer mais escrever
    notifyAll();
    System.out.println ("le.escritorSaindo("+id+")");
  }
}
```