Lista 2 – Computação Concorrente

Nome: Gabriel Almeida Mendes DRE: 117204959

Q1) Tarefa: Considerando a solução concorrente apresentada no programa acima, responda as perguntas abaixo, justificando todas as respostas:

a) Quais são as etapas principais do programa? Como ele funciona em detalhe?

As principais etapas do programa estão na sincronização por barreira e na função das threads, mais especificamente no laço for. Primeiro, o programa recebe o número de elementos do vetor de entrada e atribui esse mesmo valor ao número de threads, em seguida inicializa as variáveis globais e preenche o vetor correspondente a ela, os valores passados podem ser pré-determinados ou gerados aleatoriamente. Segundo, um for cria as threads e chama a função tarefa que é executada pelas mesmas, nessa função que ocorrerá as operações principais do programa. Como num programa concorrente todas as threads são executadas ao mesmo tempo e nesse também é usado uma variável compartilhada entre as threads, a função tarefa se utiliza de um mecanismo de sincronização para garantir que não haja condição de corrida, dessa forma a função tarefa funciona assim:

Cada thread percorrerá o laço for um número equivalente a mesma quantidade de threads para quando chegar ao condicional só passar as threads com número de id valido, as que não passarem quebram o laço e são preenchidas com os valores padrão. Para as que passaram, nas linhas seguintes estarão as linhas centrais do código, uma variável auxiliar será declarada e nela será armazenada um valor que é correspondente a uma posição do vetor que será a de número anterior daquela que, tem como referência de posição atual do vetor, um valor que o mesmo que o do id da thread atual do loop atual. Exemplo: Loop 1 da thread de id 2, seu aux será 4 porque é o valor correspondente a da posição v[1] = v=[2 – 1] = v=[id-salto], e assim por diante. A barreira que vem depois e uma barreira simples que apenas garante que todas threads tenham calculado suas variáveis auxiliares antes de prosseguirem. A próxima linha é que modificará realmente o vetor e realizar a soma parcial, fará somando a posição atual do vetor, identificado pelo id da thread atual, e somá-lo com o valor anterior, representado pela variável aux. Exemplo: Loop 1 da thread de id 1, terá o valor posição vetor[1] modificada para 5 porque é o resultado de vetor[1] = 1 + vetor[1] = 1 + 4. A barreira seguinte garante que todas as threads alterem o vetor antes de passar para o próximo laço. O laços continuaram acontecendo até que todas as posições sejam alteradas pelo resultado da soma parcial correta.

(b) A solução apresentada está correta? O resultado dela equivale ao resultado que seria obtido com algoritmo sequencial apresentado?

A solução está correta e se fosse feito um algoritmo sequencial este teria obtido o mesmo resultado. Porque o que a solução concorrente faz e o que a torna, basicamente, mais vantajosa nesse caso, é que uma thread reutiliza contas realizadas por outras threads. Com isso há mais ganho de tempo e performance do programa concorrente em relação a solução sequencial.

(c) Por que são necessárias duas chamadas de sincronização coletiva (implementada pela função *barreira()*)? Elas poderiam ser substituídas por apenas uma?

Não, elas não poderiam ser substituídas por apenas uma, porque ambas possuem objetivos diferentes. A primeira barreira garante que todas as threads tenham calculado suas variáveis auxiliares antes de seguirem para as alterações do vetor. Já a segunda barreira garante que todas as threads tenham preenchido o vetor antes de seguirem para o próximo laço do for.

Q2) Tarefa: Implemente uma thread adicional T2 para necessariamente imprimir na tela o valor da variável contador sempre que ele for múltiplo de 100 (indicando que a função˜FazAlgo foi executada por mais 100 vezes). Crie outras variáveis globais e altere o código de T1, caso necessário. Comente seu código.

```
void FazAlgo(long int n){}
void *T1 (void *arg) {
  while(1) {
    FazAlgo(contador);
    pthread_mutex_lock(&mutex); //variavel mutex acionada
    contador++; //sessao critica
    pthread_cond_wait(&cond, &mutex); //impedia a 1º thread de proseguir até que seja sin
aliza pela thread 2
    pthread_mutex_unlock(&mutex); //variavel mutex liberada
  }
}
void *T2(){
  while (1){
    pthread_mutex_lock(&mutex); //variavel mutex acionada
    /* Sessao critica com a variavel */
    if (contador % 100 == 0){
      if (contador != antes){
        antes = contador;
        printf("%lld\n", contador);
      }
    }
    pthread_mutex_unlock(&mutex); //variavel mutex liberada
    pthread_cond_signal(&cond); //sinaliza para a 1º thread que ela pode prosseguir, já que
a segunda ja foi verificada e execuada
  }
}
```

Q3) Tarefa: (a) Descreva como essa implementação do *pool de threads* funciona. (b) Identifique o erro no código e mostre corrigi-lo. Justifique suas respostas.

Na classe privada MyPoolThreads, é que reside a parte central para o funcionamento do programa, pois é nela que temos o método que usamos para executar as tarefas nas threads (o método *run*). Nessa classe temos uma variável r to tipo runnable. Em seguida temos um laço que, com a ajuda do synhronized para a implementação de mecanismos de sincronização, vamos primeiramente verificar se nossa lista que está vazia e o shutdown desligado (false). Enquanto for verdadeiro faremos com que a thread pare de executar até receber o notify, logo depois que um elemento do tipo Runnable seja colocado na lista e a thread enfim ter alguma tarefa para realizar.

Na próxima etapa, passamos por um if em que verificaremos se queue está vazia e se shutdown foi modificada pra true. Em caso de verdade, apenas damos um return, indicando que as tarefas da fila acabaram. Em caso de falso, pegamos o primeiro elemento de queue, o removemos da lista e então retornamos para a variável runnable r. Por último, a variável vai ser envolvida por try/catch, executada pelo método .run. É assim que cada thread executa uma tarefa.