Trabalho 2 – QuickSort – Computação Concorrente

Tiago Santos Martins de Macedo

DRE 116022689

Projeto

Para implementarmos um *pool* de tarefas a serem executadas por uma quantidade fixa de threads, devemos, inicialmente, determinar quais dados serão globais, e quais estruturas auxiliáres serão necessárias.

Analisemos, inicialmente, como o QuickSort funciona:

```
int[] A = {16, 18, 0, 33 /*...*/}; // array global a ser ordenado

// Função de alto nível, responsável pela parte recursiva:
void quicksort(int inicio, int fim) {
    int p = particao(inicio, fim);
    quicksort(inicio, p);
    quicksort(p, fim);
}

// Função de baixo nível, responsável pela ordenação em si:
int particao(int inicio, int fim) {
    if (fim - inicio == 1)
        return inicio;
    // elemento aleatório do array
    int p = rand() % (fim - inicio);

    int i=inicio, j=fim-1;
    while (1) {
        while (A[i] < A[p]) i++;
        while (A[j] > A[p]) j--;
        if (i < j)
            troca(A, i, j); // troca os elementos de lugar
        else
            break;
    }
    return p;
}
```

Podemos ver que o ordenamento efetivamente ocorre na função `partição`; a função `quicksort` serve simplemente para dividir o array em duas partes e, recursivamente, chamar `quicksort` para cada uma dessas partes. Ao invés disso, uma forma concorrente de implementar tal algoritmo seria adicionar referências a cada metade do array a uma "fila de partes de array a serem processadas". Assim, teríamos a seguinte versão de `quicksort`:

```
// ...
pedaco P = proximoPedaco(); // retorna ponteiro para próximo pedaço
quicksort(P);
// ...
void quicksort(pedaco P) {
    int inicio = P->inicio;
    int fim = P->fim;
    int p = particao(inicio, fim);
    pushFila(inicio, p); // adiciona pedaço à fila
    pushFila(p, fim); // adiciona outro pedaço à fila
    removePedaco(P); // tira pedaço da fila
}
```

Assim, uma quantidade fixa de threads percorrerá a fila, retirando uma referência ao pedaço do array a ser processado, processando-o e adicionando os dois sub-pedaços à fila. Note que o array em si continua sendo um só; a lista de pedaços consiste apenas em referências a inícios e fins de partes desse array. Não há risco de condição de corrida, pois cada thread tem acesso exclusivo ao pedaço que recebe; só após seu encerramento é que outras threads poderão agir sobre seus sub-pedaços.

Antes da inicialização das threads, o pedaço inicial será adicionado à fila pela própria função `main`. Quando não houverem mais pedaços para serem processados, o algoritmo encerra.

Parece um bom plano. Mas onde estão as possíveis condições de corrida?

```
pedaco P = proximoPedaco(); // <--- lê e altera dados globais
quicksort(P);
// ...
void quicksort(pedaco P) {
    int inicio = P->inicio;
    int fim = P->fim;
    int p = particao(inicio, fim);
    pushFila(inicio, p); // <--- lê e altera dados globais
    pushFila(p, fim); // <--- lê e altera dados globais
    removePedaco(P); // <--- altera dados globais
}
```

Agora, temos que implementar a fila e os pedaços.

A fila será, naturalmente, global. Como não sabemos de quantos elementos vamos precisar, utilizaremos uma lista duplamente encadeada. Ademais, haverão dois ponteiros globais: o primeiro, `cabeca`, apontando para o começo da fila (para fazermos `proximoPedaço`), e o segundo, `rabo`, apontando para o fim (para fazermos `pushFila`). A lista é *duplamente* encadeada para que `removePedaco` possa retirar um elemento da lista, fazendo o anterior apontar para o próximo, rapidamente.

Do ponto de vista das threads, o que deve ser feito é simples:

```
void* thread(void* arg) {
    while (1) {
        if (cabeca == NULL) pthread_exit(NULL); // <--- lê dados globais
        pedaco P = proximoPedaco();
        quicksort(P);
    }
}
```

Quanto ao tipo `pedaco`, sabemos que ele deve conter as seguintes informações:

- inteiro que marca o início do pedaço,
- inteiro que marca o fim do pedaço,
- ponteiro para próximo pedaço da lista,
- ponteiro para pedaço anterior na lista.

É trivial implementar um struct com essas informações.

```
typedef struct _PEDACO {
    int inicio;
    int fim;
    struct _PEDACO* prox;
    struct _PEDACO* ant;
} PEDACO *pedaco;
```

Para tratarmos as condições de corrida, vamos utilizar locks simples nas funções `proximoPedaco`, `removePedaco` e `removeFila` -- três funções que operam sobre a fila, que é global. Depois disso, resta apenas um caso a tratar: quando a thread faz `while (cabeca == NULL)`. Basta circundar esta sessão com um lock. Haverão dois mutexes: um para regular o acesso à variável `cabeca`, e outro para regular a variável `rabo`. A implementação de todos esses locks pode ser vista no código.

Obsevação: no código real, os nomes de variáveis estão escritos em inglês, afim de facilitar a vida do programador, dado que as palavras-chaves de C e os nomes de função da livraria pthread também estão em inglês.

Na prática

Um problema que ocorreu ao longo da implementação foi que, ao rodar o programa com mais de uma thread, a primeira delas consumia o primeiro elemento da lista, fazendo com que esta ficasse temporariamente vazia. A ideia é que, depois que essa thread executasse a função `quicksort`, dois novos elementos seriam adicionados à lista, para serem consumidos pelas outras threads. Na prática, a segunda thread entrava em execução exatamente nesse momento, se deparava com uma thread vazia, achava que o algoritmo havia encerrado, e terminava.

Para resolver esse problema, criei uma variável responsável por regular a *quantidade de partes do array na fila ou sendo processadas*, ou seja, a quantidade de elementos na fila somada à quantidade de elementos sendo processados pela função `quicksort`. Essa variável, é claro, também requer seu mutex.

Testes

Para testes, usei o arquivo: `test`, que contém uma sequência aleatória de 1000000 números entre 0 e 1000000, podendo haver repetições.