Distribuição de trabalhos: pthreads e mutex

Uma função que realiza um longo trabalho tipicamente atribui um identificador de trabalho para ele. Imagine que o método `do_all_work` realize todo o trabalho e a função `work` realiza o processamento de um trabalho, dado um identificador único. Podemos definir da seguinte forma:

```
void do_all_work(int n)
{
    for (int i = 0; i < n; i++)
    {
        work(i);
    }
    return;
}
```

O código acima utiliza a função `work` para processar sequencialmente `n` trabalhos. Nesta questão, você deve paralelizar a função `do_all_work` com várias threads.

Implementação

Para implementar, você deve definir no seu código:

```
int count;
pthread_mutex_t mut;
struct thread_arg {
    int vezes
};
void *thread_func(void *arg)
{
    ...
}
```

Que são, respectivamente, um contador global chamado `count`, que realiza a contagem do número de requisições atendidas, um mutex chamado `mut`, que protege o contador `count`. Como o contador é uma região de memória compartilhada entre as threads, toda operação de escrita em count deve estar protegida pelo mutex. Além disso, as threads são iniciadas em uma função chamada `thread_func`. A função recebe como argumento um ponteiro para a `struct thread_arg`, que contém dentro delas apenas um único inteiro chamado `vezes`.

No seu código, você deve definir a `struct thread_arg`, implementar a função `thread_func`, definir e **inicializar** corretamente as variáveis `count` e `mut`. Não mude o nome delas.

Além disso, a função `work` pode demorar muito tempo para processar o trabalho com `id`. Por isso, essa função não pode ser chamada com o mutex obtido.

```
int work(int id);
```

Você pode colocar o trecho de código com o cabeçalho da função `work` definido acima para evitar avisos e erros de compilação.

O problema

Nesta questão, você deve paralelizar a função `do_all_work` em várias threads. Cada thread deve chamar a função `work`, passando um argumento de id único.

Para isso, você deve implementar a função `thread_func`. Esta função recebe como argumento um ponteiro da `struct thread_arg*`, que contém o número de vezes que cada a thread deve chamar a função `work`. Nesta questão, a struct de argumentos contém apenas o número de vezes que cada thread deve trabalhar. **Não** está disponível um identificador único para cada thread (id 0 para primeira thread, id 1 para a segunda, etc. . .).

Para controlar o trabalho, o contador global count deve ser incrementado toda vez que `work` for chamada. Não se pode passar mais de uma vez o mesmo valor de id para `work` e todos os ids devem ser passados.

Sincronização

As threads **não** devem ser sincronizadas para chamar `work`. Não é necessário que a primeira thread chame work as primeiras n-vezes. Queremos explorar o pararelismo: é seguro executar em paralelo várias chamadas da função `work` com ids diferentes.

Restrições

Seu código não deve incluir a implementação da função `work`, nem da função main.

Entrada

Não há dados de entrada para serem lidos.

Saída

Não há dados de saída para serem impressos. Recomenda-se imprimir apenas para debug.

Exemplos

Para testar o seu código, você deve usar a criativade e implementar localmente uma função main e um método `work`. Por exemplo, se a função main criar 5 threads, passando o argumento "vezes" como 1 para cada uma das threads. Podemos fazer a função `work` imprimir o id. Dessa forma, deve-se imprimir 0, 1, 2, 3, 4 na tela (Não necessariamente nessa ordem!).

Author: Daniel Sundfeld <daniel.sundfeld@unb.br>