Insper

25 - POSIX Threads II

Igor Montagner

Parte 0 - revisão da última aula

Se você já fez a parte 4, faça os exercícios abaixo. Se não, você tem 20 minutos para fazê-la.

Exercício: Faça uma versão sequencial do seu programa e meça o tempo de execução. Use o programa `entrada.py` para gerar entradas grandes. Escreva abaixo os tempos e tente explicá-los.

Exercício: Modifique seu programa para fazer a soma dos logs do vetor. Faça o experimento acima de novo e reporte abaixo. Houve melhora?

Exercício: adapte seu programa para aceitar a criação de um número arbitrário de threads (pode ser somente potência de 2). Faça de novo o experimento. É melhor adicionar mais threads? Até quando?

Parte 1 - proteção de dados usando `mutex`

Vamos simular o compartilhamento de dados em aplicações usando nosso exemplo da soma de vetores. Para facilitar, crie uma cópia de seu exercício. Iremos comparar as duas versões.

Exercício: Crie uma variável global `double soma` e modifique sua tarefa para que ela faça a soma diretamente na variável global. Tudo continua funcionando? Você consegue explicar por que?

Vamos agora trabalhar agora para corrigir este erro! Lembrando da aula, as operações possíveis são as seguintes:

- `lock` - se tiver destravado, trava e continua; se não estiver espera.
- `unlock` - se tiver a trava, a destrava e permite que outras tarefas travem.

Não existe garantia de ordem! Por enquanto isto não será um problema, mas na próxima aula veremos casos em que isto é um problema.

> **!** Você pode precisar instalar o pacote `manpages-posix-dev` para obter essas páginas do manual.

Exercício: Identifique no seu código quais linhas compõe a região crítica e onde deveriam estar as diretivas `lock` e `unlock`.

Exercício: O mutex precisa ser criado e inicializado. Onde isto deve ser feito?

Exercício: Consulte os seguintes manuais e use-os para consertar seu programa usando mutexes.

- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Exercício: Seu código arrumado funciona? Meça o tempo e compare com o original (2 threads).

Como vemos no exemplo acima, usar primitivas de sincronização é caro! O modelo que fizemos originalmente (na parte 4 do roteiro anterior) é chamado de *fork-join* e será foco da primeira parte da disciplina de SuperComputação.

Parte 2 - desafios

Esta parte contém alguns desafios de programação com threads.

1. Ordenação paralela: a função `qsort` pode ser usada para ordenar dados em *C*. Faça uma versão que ordena duas partes de um vetor (uma em cada thread) e junte os resultados na thread principal. Meça o tempo que levou.
2. Busca em arquivos: leia um arquivo de texto por completo em uma região de memória e faça a busca em paralelo por termos neste arquivo.