Biblioteca mthread

Lucas Perin - 229712 Lucas Weigel - 229786

4 de Maio de 2015

Conteúdo

Resumo

Este trabalho tem como fim produzir uma biblioteca de threads na linguagem de programação C. Ela será do tipo n:1, onde várias threads dividirão apenas um fluxo de execução real no processador.

Capítulo 1

Funções

A biblioteca produzida contém 6 funções que podem ser usadas para gerenciar um ambiente multithreading. Como a biblioteca segue o modelo N:1, ela não terá um ambiente multithreading real, mas simulará um. Sendo assim a bibilioteca mthread gerada apenas poderia ser utilizada na prática com algum ganho real em processadores monocore.

1.1 mcreate

A função mcreate da biblioteca é a função responsável por criar o novo fluxo de execução (ou thread), ela aloca a memória necessária e coloca o novo fluxo de execução no estado pronto. Para assim que a nova thread chegar na vez, a mesma possa ser colocada em execução. Segundo nossos testes essa função funciona como deveria.

1.2 myield

A função myield é responsável por liberar a thread que está sendo executada atualmente, e coloca-lá de volta ao estado de pronta. Dando assim, lugar a alguma outra thread de prioridade igual ou superior a ela. Se caso ela for a única thread na sua prioridade e não houver ninguém com prioridade maior do que ela, a mesma volta para o fluxo de execução. Segundo nossos testes a função myield funciona como deveria.

1.3 mwait

A função mwait é reponsável por fazer a thread no fluxo de execução atual esperar pela thread com tid fornecido acabar. Para então voltar ao estado pronto e continuar sua execução. Se caso a thread pela qual ela deve esperar não existir ou já tiver acabado sua execução a função retorna imediatamente sem fazer nada e continua executando a thread original (que chamou a mwait). Segundo nossos testes a função mwait funciona como deveria.

1.4 mmutex-init

A função mmutex-init tem como reponsabilidade principal inicializar a estrutura previamente alocada do tipo mmutex. Ela apenas serve para que futuras funções utilizando a estrutura dada não recebam valores não inicializados e provavelmente equivocados. Segundo nossos testes a função mmutex-init funciona como deveria.

1.5 mlock

A função mlock recebe uma estrutura previamente alocada e inicializada, e tenta bloquea-la, caso ela ja esteja bloqueada a thread atual sai de execução e entra na fila da estrutura para ser desbloqueada. Caso ela ainda não esteja bloqueada, a função apenas bloqueia a estrutura e continua a execução da thread atual. Segundo nossos testes a função mlock funciona como deveria.

1.6 munlock

Esta função recebe uma estrutura do tipo mmutex que foi previamente bloqueada pela mesma thread que está em execução no momento. Caso a estrutura não tenha nenhuma thread na fila para bloquea-la, a thread atual (que chamou a função munlock) apenas desbloqueia a estrutura e continua sua execução. Mas caso a estrutura tenha alguma thread na fila de bloqueio, a estrutura continua bloqueada e poe a primeira thread da fila no estado pronto, para que ela possa ser chamada na sua vez. Segundo nossos testes a função munlock funciona como deveria.

Capítulo 2

Testes

Para os testes, foi utilizada a função assert(), que testa se o valor passado é igual a 1 ou true, caso não seja, ela termina o programa e indica onde ocorreu o problema. É uma função bastante útil para tanto testar quanto saber onde o problema ocorreu.

Alem disso todas as funções imprimem 'SUCCESS!' e retornam 0 caso elas tenham executado com sucesso. Caso contrario se o erro ocorrer em um dos assert, o mesmo imprime onde o erro ocorreu, e caso um segmentation fault ou algo parecido ocorra o processo retorna um número diferente de 0, e provavelmente não imprime a frase 'SUCCESS!'.

Temos também além de testes das primitivas da biblioteca alguns testes das listas e vetores utilizados para guardas as estruturas de dados das threads. Fizemos eles pois as listas e vetores são provavelmente a parte do código que é mais passível de erros de programação, assim garantimos que essa parte estará funcionando de maneira adequada.

2.1 test-mcreate

Este teste tem como objetivo testar a função mcreate, ela apenas cria 'x' threads e testa se a função não retornou erro em nenhuma delas. A princípio esse teste não garante muito, apenas que a função mcreate não está tendo problemas na criação de threads.

2.2 test-mlist-exist-tid

Este teste garante que qualquer thread que eu incluir na lista, constará como presente nas buscas seguintes.

2.3 test-mlist-pop-first

Este teste garante que a ordem das threads que foram colocadas seja a mesma na hora de retira-las pelo início.

2.4 test-mlist-pop-tid

Este teste garante que eu possa retirar qualquer thread do meio da lista sem causar problemas de ponteiros ou algo do tipo. Mesmo que a biblioteca não utilize essa função da lista, como ela existe, existe um teste para ela.

2.5 test-mlist-pop-tid-random

Este teste é um complemento do 'test-mlist-pop-tid', ele garante que possamos retirar qualquer thread de qualquer lugar do vetor sem problemas. O teste é feito de forma que ele retire os elementos da lista de forma aleatória, e repita o processo uma certa quantidade de vezes, garantindo assim que com todas as ordens de retirada a lista funciona.

2.6 test-mthread

Este teste é provavelmente o segundo teste mais geral, ele testa o funcionamento das 3 funções que lidam com threads (mcreate, myield, mwait), é um teste simples que apenas testa se elas estão executando corretamente.

2.7 test-mutex

Este é provavelmente o teste mais completo, ele testa praticamente tudo, se este teste rodar com sucesso, provavelmente todos os outros irão também. Ele testa o funcionamento do mutex dentro de várias threads lutando para entrar na parte de código crítica. Muitos erros foram encontrados no nosso código por causa desse teste, então eu diria que ele é o mais importante da nossa seleção de testes.

2.8 test-mvector-exist-tid

Esta é uma variação do 'test-mlist-exist-tid', ele faz basicamente a mesma coisa, mas com a estrutura de vetor. Ele apenas testa se todos os elementos

inseridos no vetor, constam depois na busca por eles.

2.9 test-mvector-pop-tid

Esta é uma variação do 'test-mlist-pop-tid', que também faz a mesma coisa do original, mas com a estrutura vetor. Ele testa se depois de inserir e retirar o elemento do vetor, o mesmo continua intacto.

2.10 test-mwait

Este é um dos testes primitivos, ele testa de forma bastante simples a função mwait. Apenas testando se após a execução do mwait, todas as threads executaram corretamente.

2.11 test-myield

Este é outro dos testes primitivos, ele testa se após um myield, alguma thread foi executada (como as threads foram criadas com prioridade alta, com certeza alguma deve ser executada).

Capítulo 3

Dificuldades

3.1 Listas e Vetores

Com toda a certeza a maior dificuldade encontrada foi fazer as listas e vetores do código funcionarem 100%. Sempre havia algum problema em algum lugar e levava muito tempo para debugar e achar o erro. Por isso que foram criados os testes de lista e vetor, para facilitar a solução de bugs nessas estruturas. Depois de começar a usar os testes o projeto andou com muito mais facilidade, pois sabíamos por onde começar a procurar por problemas.

3.2 Performance

O projeto também teve um "problema"de performance, que em alguns casos a biblioteca era muito lenta. Isso se devia ao fato de no início, ser usado apenas listas, e para busca de alguma thread na lista ou ainda para retirada de uma thread do meio da lista, era um processo muito lento, e com uma chance de um bug incomodar depois muito grande. Por isso foi decidido que seria usado vetores ao invés da lista de bloqueado, pois nela não importava a ordem de entrada das threads, além disso usamos vetores auxiliares junto com as listas, para que as buscas pudessem ser feitas instantaneamente, ou seja, complexidade O(1). Depois de essa mudança concluída chegamos a ter ganhos de até duas mil vezes, dependendo do número de threads usado no teste.