

Programando em Paralelo: Pthreads

Introdução

O objetivo principal desta atividade é ganhar experiência com paralelismo multicore usando a biblioteca POSIX Pthreads padrão, incluindo exposição a conceitos de multithreading, como mutexes e condições. O objetivo secundário é ganhar experiência na implementação de um modelo de fila de tarefas mestre/trabalhador de computação paralela.

Descrição

O programa lê uma lista de "tarefas" de um arquivo. Cada tarefa consiste em um código de caractere que indica uma ação e um número. O código do caractere pode ser um "p" (para "processar") ou "e" (para "esperar"). O arquivo de entrada simula várias cargas de trabalho entrando em um sistema de multiprocessamento. Em um sistema real, as ações "p" (as tarefas) provavelmente seriam chamadas para rotinas computacionais. Para nossos propósitos, "processar" uma tarefa com número n significa apenas esperar n segundos usando a função sleep e então atualizar algumas variáveis agregadas globais (soma, contagem ímpar, mínimo e máximo). A ação "e" fornece uma maneira de simular uma pausa nas tarefas de entrada.

Por exemplo, o script a seguir simula uma tarefa inicial de um segundo entrando no sistema, seguida por um atraso de dois segundos. Após o atraso, uma tarefa de dois segundos entra no sistema seguida por uma tarefa de três segundos.

```
p 1
e 2
p 2
p 3
```

Usando um sistema de processamento puramente serial (conforme implementado em sum.c contido no Buraco Negro), o cenário acima levará oito segundos para terminar, conforme a seguir.

t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	

p 1	e 2		p 2		p 3				

									versão sequencial

A saída final deve corresponder ao seguinte (soma, # ímpar, mínimo, máximo):

```
6 2 1 3
```

Neste projeto, você vai estender este programa para aproveitar as vantagens de uma CPU multicore usando um modelo de fila de tarefas. Nesse modelo, o programa principal gera um número definido de threads de trabalho. Você deve ler o número de threads da linha de comando como um novo segundo parâmetro. O programa principal e os threads de trabalho se comunicam usando uma fila de tarefas para controlar as tarefas que ainda precisam ser processadas.

Observe que se permitíssemos que a primeira tarefa fosse processada durante a espera, poderíamos reduzir o tempo em um único segundo:

t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	

e 2		ocioso						mestre

p 1	ocioso	p 2		p 3				único trabalhador

Isso pode ser obtido dividindo o trabalho de processamento real em um thread de trabalho que pode funcionar paralelamente ao thread mestre original. Isso permite que o thread mestre se concentre em receber trabalhos, enquanto o thread de trabalho se concentra em fazer o trabalho real. No cenário acima, a primeira tarefa chega em $t = 0$, enquanto as duas segundas tarefas chegam simultaneamente em $t = 2$.

Como nossas tarefas são independentes, a situação pode ser melhorada ainda mais com a adição de mais threads de trabalho, supondo que tenhamos núcleos de CPU físicos suficientes para tirar proveito deles. No exemplo acima, as duas últimas tarefas podem ser executadas simultaneamente para economizar dois segundos adicionais:

t=0	t=1	t=2	t=3	t=4	t=5	

e 2		ocioso				mestre

p 1	ocioso	p 2		ocioso		trabalhador 1

ocioso		p 3				trabalhador 2

Seu programa deve funcionar da seguinte maneira. No início da execução, o thread mestre gera um número definido de threads de trabalho (fornecido por um parâmetro de linha de comando). Os threads de trabalho estão inativos no início. Uma vez que os trabalhadores tenham sido totalmente inicializados, o mestre então começa a lidar com as tarefas do arquivo de entrada adicionando-os a uma fila de tarefas, ativando um thread de trabalho ocioso (se houver) para cada tarefa. Quando um thread é ativado, eles começam a puxar tarefas da fila e processá-las. Se a fila ficar sem tarefas, o trabalhador deve bloquear novamente até ser acordado pelo mestre. Se o mestre encontrar um comando "e" (esperar), ele aguardará o número determinado de segundos antes de continuar no arquivo de entrada. Depois que todas as tarefas foram adicionadas à fila, o mestre espera que a fila se esgote, ativando os threads inativos conforme necessário para ajudar. Quando a fila está vazia, o mestre espera que os trabalhadores não ociosos terminem e, em seguida, define um sinalizador global para indicar que todo o programa foi concluído, reativando todos os threads de trabalho para que possam ser encerrados. O mestre então limpa tudo e sai.

Para implementar o sistema acima, você deve usar threads de Pthread, mutexes e condições conforme abordado na aula. Seu programa deve considerar o número de threads de trabalho como um segundo parâmetro de linha de comando; o desempenho em cargas de trabalho paralelizáveis deve ser escalonado linearmente com o número de threads. Você deve usar pelo menos um mutex (embora provavelmente precise de mais de um) e pelo menos uma condição. Você também deve manter uma estrutura de dados da fila de tarefas explícita.

Você NÃO PODE usar semáforos para esta tarefa. Embora possa haver uma solução para esse problema usando semáforos, um dos principais objetivos deste projeto é ganhar experiência com mutexes e condições.

Dicas:

- Adicione uma nova função de thread de trabalho para a maior parte da sincronização do lado do trabalhador.
- Implemente a fila de tarefas como uma lista ligada individualmente.
- Use um mutex para proteger as variáveis agregadas globais.
- Use um mutex para proteger a fila de tarefas.
- Use uma condição para bloquear e despertar threads de trabalho e considere o uso de seu mutex associado para proteger um contador de "thread inativo".
- Declare variáveis que você não deseja que o compilador otimize usando a palavra-chave `volatile`.
- Não se esqueça dos loops `do/while`!
- Está tudo bem para o thread mestre utilizar espera-ocupada.
- Compile com `-O0` para depurar e mude para `-O2` depois de ter certeza de que o básico está funcionando.
- Crie vários arquivos de entrada de teste e um script para executar automaticamente (e cronometrar) todos eles.
- Tente remover a chamada para "dormir" na função de atualização para comprimir os tempos de trabalho e verificar as condições de corrida.
- Não exagere na engenharia! Quanto menor o código melhor.

Critérios de Grade

Nota	Descrição	Requisitos
9,5 – 10,0	Excepcional	<ul style="list-style-type: none">- Speedup linear com vários threads de trabalho.- Tempos de execução consistentemente ideais.- Resultados corretos em todos os casos de teste em todos os níveis de otimização.- Uso correto de todas as construções Pthread.- Falta de qualquer potencial condição de corrida.- Código limpo com estilo totalmente consistente.- Documentação perspicaz.- Relatório discute perspectivas futuras.- Todas as opções abaixo.
8,0 - 9,5	Excelente	<ul style="list-style-type: none">- Speedup com vários threads de trabalho.- Termina em todos os casos de teste em todos os níveis de otimização.- Implementação correta de uma fila de tarefas.- Uso correto de vários mutexes Pthread.- Uso correto da condição Pthread.- Estilo consistente e razoável.- Sem vazamentos de memória.- Documentação útil e consistente.- Relatório discute pontos relevantes sobre desempenho e speedup.- Todas as opções abaixo.
4,1 – 7,9	Bom	<ul style="list-style-type: none">- Compila sem precisar de modificações.- Resultados corretos com um único thread de trabalho.- Uso correto de pelo menos um mutex Pthread.- Uso de pelo menos uma condição Pthread.- Não usa semáforos.- Estilo legível.- Quantidade mínima de documentação.- Relatório prove evidências de uma análise adequada.- Todas as opções abaixo.
1 – 4,0	Deficiente	<ul style="list-style-type: none">- Algumas evidências de uma tentativa de boa fé.
0	Inaceitável	<ul style="list-style-type: none">- Não compila.- Erro de execução.- Código não parametrizável.- Código sem uso de construções Pthreads.

Relatório

O relatório deve ser no formato IEEE Latina, conter no máximo 4 páginas e ter NECESSARIAMENTE as seguintes seções:

- Introdução
- Descrição do Problema
- Resultados e Discussão
- Conclusões
- Referências

Submissão

Você deve copiar sum.c em um novo arquivo chamado par_sum.c e modificá-lo para implementar o sistema mestre/trabalhador descrito acima. Seu programa modificado deve aceitar DOIS parâmetros de linha de comando em vez de um: 1) o nome do arquivo de entrada e 2) o número de threads de trabalho. Seu programa deve ser compilado em sistemas Linux compatíveis com POSIX usando o seguinte Makefile:

```
default: sum par_sum
```

```
sum: sum.c
    gcc -g -O2 --std=c99 -Wall -o sum sum.c
```

```
par_sum: par_sum.c
    gcc -g -O2 --std=c99 -Wall -o par_sum par_sum.c -lpthread
```

```
clean:
    rm -f sum par_sum
```

Você deve submeter um pacote zip contendo: (1) makefile, (2) par_sum.c, (3) fontes do relatório em latex e (4) pdf do relatório.