Tarefa 5: Concorrência com threads

17 de junho de 2025

1 Implementação

Foi implementado um programa em linguagem C que cria duas threads que incrementam concorrentemente uma variável global compartilhada:

```c
#include <stdio.h>
#include <stdint.h>
#include <pthread.h>

uint64_t valor = 0;

void* thread(void* arg) {
    size_t i = 1000000;
    while (i--) {
        valor++;
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // Criar duas threads
    pthread_create(&t1, NULL, thread, NULL);
    pthread_create(&t2, NULL, thread, NULL);

    // Aguardar as threads terminarem
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    // Imprimir o resultado
    printf("Valor final: %lu\n", valor);

    return 0;
}
```

O programa foi compilado de duas formas diferentes: uma com as configurações padrão do GCC e outra com otimizações habilitadas (`-O3`).

2 Análise do Comportamento

2.1 Resultado Sem Otimizações

Ao executar o programa compilado com as configurações padrão do GCC, o resultado final é consistentemente menor que 2 milhões, variando em cada execução.

2.2 Resultado Com Otimizações

Ao compilar o programa com a flag `-O3`, o resultado é sempre exatamente 2 milhões.

3 Respostas às Questões

3.1 Por que o resultado não é 2 milhões quando compilado com gcc nas configurações padrões?

O resultado não é 2 milhões devido a uma condição de corrida (*race condition*) entre as duas threads. A operação `valor++` não é atômica e, na realidade, é composta por três operações distintas:

1. Ler o valor atual da variável da memória para um registrador
2. Incrementar o valor no registrador
3. Escrever o novo valor de volta na memória

Sem otimizações, as duas threads executam estas três operações concorrentemente, o que pode levar a cenários como:

1. Thread 1 lê o valor atual (por exemplo, 1000)
2. Thread 2 lê o mesmo valor atual (1000)
3. Thread 1 incrementa para 1001
4. Thread 2 incrementa para 1001
5. Thread 1 escreve 1001
6. Thread 2 escreve 1001

Neste cenário, embora duas operações de incremento tenham sido realizadas, o valor final é incrementado apenas uma vez. Este fenômeno, conhecido como condição de corrida, ocorre repetidamente durante a execução, resultando em um valor final menor que 2 milhões.

Analisando o código assembly gerado pela compilação sem otimizações, podemos ver claramente essas três operações:

```
1 117b: 48 8b 05 b6 2e 00 00  mov    0x2eb6(%rip),%rax        # 4038 <
    valor>
2 1182: 48 83 c0 01           add    $0x1,%rax
3 1186: 48 89 05 ab 2e 00 00  mov    %rax,0x2eab(%rip)        # 4038 <
    valor>
```

Estas três instruções correspondem exatamente às três operações descritas: a primeira instrução lê o valor atual para o registrador RAX, a segunda instrução incrementa o valor no registrador, e a terceira instrução escreve o novo valor de volta à memória. Como as duas threads executam essas mesmas instruções concorrentemente, a condição de corrida ocorre frequentemente.

3.2 Por que o resultado é 2 milhões quando habilitamos as otimizações do gcc?

Com as otimizações habilitadas (`-O3`), o compilador GCC transforma significativamente o código gerado. Ao analisar o código assembly produzido, observamos que o compilador realiza otimizações agressivas:

1. O compilador percebe que o laço `while (i--)` está simplesmente incrementando `valor` um milhão de vezes
2. Em vez de executar o incremento em um laço, o compilador substitui todo o código da função por uma única operação que adiciona 1.000.000 diretamente à variável `valor`
3. Esta única operação é atômica do ponto de vista da função, resultando em exatamente 2 milhões após ambas as threads serem executadas

Essencialmente, com a otimização, cada thread efetivamente executa `valor += 1000000` como uma operação única, eliminando o problema de concorrência presente na versão sem otimização.

Analisando o código assembly gerado com otimizações, podemos ver esta transformação claramente:

```
0000000000001200 <thread>:
    1200: 48 81 05 2d 2e 00 00  addq   $0xf4240,0x2e2d(%rip)        #
        4038 <valor>
    1207: 40 42 0f 00
    120b: 31 c0                 xor    %eax,%eax
    120d: c3                    ret
```

Observe que toda a função `thread` foi reduzida a apenas uma única instrução `addq` que adiciona diretamente o valor hexadecimal `$0xf4240` (1.000.000 em decimal) à variável `valor`. Isso elimina completamente o laço e reduz significativamente a janela de oportunidade para condições de corrida.

3.3 Com as otimizações habilitadas, seria possível o valor final ser menos de 2 milhões? Por quê?

Sim, ainda seria teoricamente possível obter um valor menor que 2 milhões mesmo com as otimizações habilitadas. Embora o compilador tenha otimizado o código para que cada thread realize uma única operação `valor += 1000000`, esta operação ainda não é atômica no nível do hardware quando se trata de múltiplas threads.

A operação otimizada ainda segue o mesmo padrão básico de leitura-modificação-escrita:

1. Ler o valor atual de `valor`
2. Adicionar 1.000.000
3. Escrever o resultado de volta

Se a segunda thread ler o valor inicial antes que a primeira thread tenha escrito seu resultado, ainda ocorrerá uma condição de corrida. No entanto, a probabilidade disso acontecer é drasticamente reduzida em comparação com a versão sem otimização, pois agora há apenas uma operação por thread em vez de um milhão.

Na prática, devido ao tempo necessário para a criação das threads e a baixa probabilidade de colisão exata no momento crítico, o resultado quase sempre será 2 milhões com otimizações, mas não há garantia absoluta.

3.4 Forma de implementar que resolve o problema da contagem errada

Uma solução para este problema seria utilizar mecanismos de sincronização, como mutexes ou operações atômicas:

```c
#include <stdio.h>
#include <stdint.h>
#include <pthread.h>

uint64_t valor = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread(void* arg) {
    size_t i = 1000000;
    while (i--) {
        pthread_mutex_lock(&mutex);
        valor++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

Alternativa usando operações atômicas:

```c
#include <stdio.h>
#include <stdint.h>
#include <pthread.h>

_Atomic uint64_t valor = 0;

void* thread(void* arg) {
    size_t i = 1000000;
    while (i--) {
        valor++;
    }
    return NULL;
}
```

Estas soluções resolvem o problema porque garantem que as operações de incremento sejam atômicas. No caso do mutex, apenas uma thread por vez pode executar o código protegido pelo mutex, eliminando a condição de corrida. As operações atômicas, por sua vez, utilizam instruções especiais de hardware que garantem atomicidade para operações básicas como incremento, também eliminando a condição de corrida.

Analisando o código assembly da versão que usa mutex, vemos a implementação da exclusão mútua:

```
119f: 48 8d 05 da 2e 00 00  lea    0x2eda(%rip),%rax        # 4080 <
   mutex>
11a6: 48 89 c7              mov    %rax,%rdi
11a9: e8 d2 fe ff ff        call   1080 <pthread_mutex_lock@plt>
```

Seguido pelo incremento e pelo desbloqueio do mutex. Esta implementação garante que apenas uma thread por vez pode executar a operação de incremento, eliminando a condição de corrida.

Na versão atômica, o compilador gera instruções especiais que utilizam recursos de hardware para garantir atomicidade nas operações. Ambas as soluções garantirão que o resultado seja consistentemente 2 milhões, independentemente das otimizações do compilador.

```
119a: f0 48 0f c1 05 95 2e  lock xadd %rax,0x2e95(%rip)        # 4038
    <valor>
```

A solução com operações atômicas tende a ser mais eficiente que a solução com mutex, especialmente para operações simples como incremento, pois evita as chamadas de sistema associadas com as operações de lock e unlock do mutex.

4 Análise dos Resultados de Execução

Ao executar as quatro versões do programa (padrão, otimizado, mutex e atômico), obtemos os seguintes resultados:

- Versão padrão (sem otimizações): `Valor final: 1011116` (varia entre execuções)
- Versão otimizada (-O3): `Valor final: 2000000` (consistente)
- Versão com mutex: `Valor final: 2000000` (consistente)
- Versão com variáveis atômicas: `Valor final: 2000000` (consistente)

Estes resultados confirmam nossa análise teórica. A versão sem otimizações sofre de condição de corrida, resultando em um valor final menor que 2 milhões. As outras versões garantem o resultado correto, cada uma por meio de um mecanismo diferente.

5 Conclusão

Este trabalho demonstra os desafios intrínsecos à programação concorrente e como o comportamento de programas com threads pode ser significativamente afetado pelas otimizações do compilador. A condição de corrida observada é um problema clássico em sistemas concorrentes, e sua resolução requer o uso adequado de mecanismos de sincronização.

A análise do código assembly gerado pelo compilador em diferentes cenários foi fundamental para compreender como o GCC transforma o código fonte e como essas transformações afetam o comportamento do programa em ambiente multithread. Com otimizações habilitadas, o compilador pode reduzir drasticamente a complexidade do código, eliminando laços e substituindo-os por operações mais diretas, como vimos na adição direta de 1.000.000 ao valor.

Além disso, este exemplo ilustra como as otimizações do compilador podem modificar drasticamente o comportamento do programa, às vezes mascarando problemas subjacentes de concorrência que permanecem teoricamente presentes, mesmo que raramente se manifestem na prática.

Para garantir a correção de programas concorrentes, é essencial compreender os mecanismos de sincronização disponíveis e aplicá-los adequadamente, independentemente do nível de otimização utilizado na compilação. O uso de mutexes ou variáveis atômicas são soluções robustas que garantem resultados corretos mesmo em ambientes de alta concorrência.