Tarefa 7: Algoritmo de Peterson

17 de junho de 2025

1 Por que usar enter_region e leave_region para estabelecer uma região crítica não funciona com compiladores modernos?

Os compiladores modernos realizam diversas otimizações de código que podem comprometer o funcionamento do algoritmo de Peterson:

- **Reordenamento de código:** Os compiladores podem alterar a ordem das instruções se acreditarem que isso não afetará a lógica do programa. No algoritmo de Peterson, a ordem exata das operações é crucial. Por exemplo, se a instrução `turn = process` for executada antes de `interested[process] = TRUE`, o algoritmo quebra.
- **Cache em registradores:** Os compiladores podem armazenar variáveis compartilhadas em registradores para melhorar o desempenho. Quando um processo atualiza uma variável compartilhada, outro processo pode ler um valor desatualizado em cache em vez do valor atual na memória.
- **Otimização de loops:** O loop de espera (`while (turn == process && interested[other] == TRUE)`) pode ser otimizado ou transformado porque aparentemente não faz nada. O compilador pode não perceber que está esperando outro processo alterar esses valores.
- **Eliminação de armazenamentos redundantes:** Sem anotações adequadas, os compiladores podem eliminar operações de memória que consideram desnecessárias, especialmente quando uma variável é escrita mas aparentemente nunca usada.

2 Como os problemas descritos na pergunta anterior podem ser resolvidos?

Essas questões de otimização do compilador podem ser resolvidas através de vários métodos:

- **Palavra-chave volatile:** Declarar variáveis compartilhadas como `volatile` informa ao compilador que essas variáveis podem mudar inesperadamente e não devem ser otimizadas:

```
1    volatile int turn;
2    volatile int interested[N];
```

- **Barreiras de memória:** Adicionar barreiras de compilador para evitar reordenamento de instruções:

```
1    interested[process] = TRUE;
2    __sync_synchronize(); // Barreira de compilador
3    turn = process;
4    __sync_synchronize(); // Barreira de compilador
```

- **Diretivas específicas do compilador:** Alguns compiladores oferecem diretivas específicas para impedir a otimização:

```
1    #pragma optimize("", off)
2    // Código do algoritmo de Peterson
3    #pragma optimize("", on)
```

- **Primitivas de sincronização padrão:** Usar ferramentas de sincronização de threads fornecidas pela linguagem, como mutexes do `<threads.h>` do C11 ou threads POSIX.

3 Com os problemas dos compiladores modernos resolvidos, qual problema ainda impede que o algoritmo de Peterson funcione em processadores modernos?

Mesmo depois de resolver os problemas de otimização do compilador, as arquiteturas de processadores modernos introduzem desafios adicionais:

- **Problemas de ordenação de memória:** Processadores modernos utilizam modelos de ordenação de memória relaxados em vez de consistência sequencial. Isso significa que as operações de memória podem não ser executadas na ordem especificada no código, mesmo sem reordenamento pelo compilador.
- **Buffers de armazenamento e caches:** Os processadores frequentemente armazenam operações em buffer para melhorar o desempenho. Uma CPU pode escrever em uma variável, mas outra CPU pode não ver essa alteração imediatamente porque:
  - A escrita está em um buffer de armazenamento
  - As CPUs têm diferentes versões em cache da memória
- **Execução fora de ordem:** Os processadores modernos executam instruções fora de ordem para otimizar o desempenho. Isso pode quebrar as suposições de sequência no algoritmo de Peterson.
- **Operações de memória não-atômicas:** Operações simples como atribuição de variáveis podem não ser atômicas em arquiteturas modernas, especialmente para variáveis que abrangem múltiplas palavras.

4 Como os problemas descritos na pergunta anterior podem ser resolvidos?

Para resolver problemas em nível de processador com o algoritmo de Peterson:

- **Barreiras de memória de hardware:** Inserir barreiras/cercas de memória para forçar a ordenação:

```
interested[process] = TRUE;
__atomic_thread_fence(__ATOMIC_SEQ_CST); // Barreira de
   hardware
turn = process;
__atomic_thread_fence(__ATOMIC_SEQ_CST); // Barreira de
   hardware
```

- **Operações atômicas:** Usar operações atômicas que garantem ordenação tanto no compilador quanto no hardware:

```
__atomic_store_n(&interested[process], TRUE,
   __ATOMIC_SEQ_CST);
__atomic_store_n(&turn, process, __ATOMIC_SEQ_CST);
```

- **Atomicidade em nível de linguagem:** Usar a biblioteca atômica do C11:

```
#include <stdatomic.h>
atomic_int turn;
atomic_int interested[N];
// Usar atomic_store, atomic_load, etc.
```

- **Mecanismos de sincronização padrão:** Substituir a implementação personalizada por primitivas de sincronização padrão e bem testadas:
  - Mutexes (`pthread_mutex_t`)
  - Semáforos (`sem_t`)
  - Locks de leitura-escrita (`pthread_rwlock_t`)

  Esses mecanismos são projetados especificamente para lidar com desafios tanto em nível de compilador quanto de processador na programação concorrente.

5 Demonstrações Práticas

Para demonstrar empiricamente os problemas e soluções discutidos anteriormente, foram implementadas cinco versões do algoritmo de Peterson:

5.1 Implementação Original

A implementação original do algoritmo de Peterson, quando compilada com otimizações:

```
/* process is 0 or 1 */
void enter_region(int process){
    /* number of the other process */
    int other;
    /* the opposite of process */
    other = 1 - process;
    /* show that you are interested */
    interested[process] = TRUE;
    /* set flag */
    turn = process;
    /* null statement */
    while (turn == process && interested[other] == TRUE);
}
```

Esta implementação apresenta comportamento não-determinístico em sistemas modernos:

- Em algumas execuções, entra em *spinlock* infinito
- Em outras, pode funcionar corretamente por acaso
- Frequentemente, falha na garantia de exclusão mútua

5.2 Desativando Otimizações do Compilador

Quando compilamos o mesmo código com a flag `-O0`, desativando otimizações:

```
$ gcc -o programa programa.c -pthread -O0
```

A probabilidade de funcionamento correto aumenta, mas ainda pode falhar devido ao reordenamento em nível de processador.

5.3 Usando Operações Atômicas

Implementação usando operações atômicas do GCC:

```
void enter_region(int process){
    int other = 1 - process;

    /* usando operação atômica */
    __atomic_store_n(&interested[process], TRUE, __ATOMIC_SEQ_CST);

    /* usando operação atômica */
    __atomic_store_n(&turn, process, __ATOMIC_SEQ_CST);
```

```
9
10      /* leitura atômica no loop de espera */
11      int other_interested, current_turn;
12      do {
13          __atomic_load(&interested[other], &other_interested,
                __ATOMIC_SEQ_CST);
14          __atomic_load(&turn, &current_turn, __ATOMIC_SEQ_CST);
15      } while (current_turn == process && other_interested == TRUE);
16  }
```

5.4 Usando Biblioteca Atômica do C11

Implementação com a biblioteca `stdatomic.h` do C11:

```
1   #include <stdatomic.h>
2
3   /* usando tipo atômico do C11 */
4   atomic_int turn;
5   atomic_int interested[N];
6
7   void enter_region(int process){
8       int other = 1 - process;
9
10      /* usando API atômica do C11 */
11      atomic_store_explicit(&interested[process], TRUE,
            memory_order_seq_cst);
12      atomic_store_explicit(&turn, process, memory_order_seq_cst);
13
14      /* leitura atômica no loop de espera */
15      while (atomic_load_explicit(&turn, memory_order_seq_cst) ==
            process &&
16              atomic_load_explicit(&interested[other],
                    memory_order_seq_cst) == TRUE);
17  }
```

5.5 Usando Barreiras de Memória

Implementação com barreiras de memória:

```
1   /* declarada como volatile */
2   volatile int turn;
3   volatile int interested[N];
4
5   void enter_region(int process){
6       int other = 1 - process;
7
8       interested[process] = TRUE;
9       /* barreira de memória */
10      __sync_synchronize();
11
12      turn = process;
```

```
    /* barreira de memória */
    __sync_synchronize();

    /* barreira dentro do loop de espera */
    while (1) {
        __sync_synchronize();
        if (!(turn == process && interested[other] == TRUE))
            break;
    }
}
```

5.6 Usando Mutex POSIX

A abordagem mais recomendada para sistemas modernos é abandonar o algoritmo de Peterson e usar primitivas de sincronização padronizadas:

```
#include <pthread.h>

/* Substituindo algoritmo por mutex */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void enter_region(int process){
    pthread_mutex_lock(&mutex);
}

void leave_region(int process){
    pthread_mutex_unlock(&mutex);
}
```

5.7 Resultados Obtidos

Testes práticos com estas implementações confirmaram:

- Versão original: exibe comportamento imprevisível e falhas frequentes
- Versão sem otimizações: funciona melhor, mas ainda pode falhar
- Versões com operações atômicas: funcionam consistentemente
- Versão com mutex POSIX: solução mais robusta e recomendada

Estes resultados experimentais validam a necessidade das técnicas discutidas anteriormente para garantir o funcionamento correto de algoritmos de exclusão mútua em sistemas modernos.