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:

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

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

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

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

```
#pragma optimize("", off)
// Código do algoritmo de Peterson
#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.