Tarefa 9: Produtor e Consumidor

Sistemas Operacionais

27 de junho de 2025

1 Problema Fundamental da Implementação

O problema fundamental na implementação fornecida do produtor e consumidor está relacionado à ausência de sincronização adequada em um cenário de concorrência. As variáveis compartilhadas entre as threads (dados[], inserir e remover) são acessadas sem mecanismos de sincronização adequados, o que pode levar aos seguintes problemas:

1.1 Race conditions (Condições de corrida)

Quando múltiplas threads tentam acessar e modificar as mesmas variáveis simultaneamente, o resultado final pode ser inconsistente. Por exemplo, dois consumidores podem tentar ler o mesmo valor simultaneamente, resultando em comportamento indefinido.

1.2 Acesso não atômico

As operações de leitura e escrita das variáveis compartilhadas não são atômicas. Isso significa que uma thread pode ser interrompida no meio da atualização dos índices inserir ou remover, levando a inconsistências.

1.3 Busy waiting ineficiente

O código utiliza busy waiting (espera ocupada) nos loops:

```
1 // No produtor
2 while (((inserir + 1) % TAMANHO) == remover);
3
```

```
// No consumidor
while (inserir == remover);
```

Isso desperdiça ciclos de CPU, pois as threads continuam executando enquanto esperam que as condições sejam satisfeitas, em vez de liberar o processador para outras tarefas.

2 Solução com Mutexes

Para resolver os problemas de sincronização, podemos usar mutexes para proteger o acesso às seções críticas:

```
pthread_mutex_t buffer_mutex = PTHREAD_MUTEX_INITIALIZER;

void *produtor(void *arg) {
    int v;
    for (v = 1;; v++) {
        int can_insert = 0;

        while (!can_insert) {
            pthread_mutex_lock(&buffer_mutex);

            if (((inserir + 1) % TAMANHO) != remover) {
                can_insert = 1;
                printf("Produzindo %d\n", v);
                dados[inserir] = v;
                inserir = (inserir + 1) % TAMANHO;
            }

            pthread_mutex_unlock(&buffer_mutex);

            if (!can_insert) {
                usleep(10000);  // Reduz uso de CPU
            }
        }

        usleep(500000);
    }
    return NULL;
}

void *consumidor(void *arg) {
    for (;;) {
        int can_consume = 0;
```

```
33
34          while (!can_consume) {
35              pthread_mutex_lock(&buffer_mutex);
36
37              if (inserir != remover) {
38                  can_consume = 1;
39                  printf("%zu: Consumindo %d\n", (size_t)arg,
    dados[remover]);
40                  remover = (remover + 1) % TAMANHO;
41              }
42
43              pthread_mutex_unlock(&buffer_mutex);
44
45              if (!can_consume) {
46                  usleep(10000);  // Reduz uso de CPU
47              }
48          }
49      }
50      return NULL;
51  }
```

Esta solução protege os acessos às variáveis compartilhadas usando mutexes, garantindo que apenas uma thread por vez execute a seção crítica. Adicionamos um sleep quando o buffer está cheio ou vazio para reduzir o consumo de CPU, mas ainda mantemos uma forma de busy waiting.

3 Problemas de Desempenho com Mutexes

A solução com mutexes resolve o problema de condições de corrida, mas apresenta problemas de desempenho:

3.1 Busy waiting modificado

Mesmo com o sleep dentro do loop, as threads ainda estão em um padrão de espera ocupada. Elas acordam periodicamente para verificar se a condição mudou, desperdiçando ciclos de CPU.

3.2 Overhead de bloqueio/desbloqueio

O mutex está sendo adquirido e liberado repetidamente em cada iteração do loop, mesmo quando a condição não permite o progresso. Isso gera um

overhead desnecessário de operações de bloqueio e desbloqueio.

3.3 Falta de sinalização

Não há um mecanismo direto para que o produtor sinalize aos consumidores quando novos dados estiverem disponíveis, ou para os consumidores sinalizarem ao produtor quando houver espaço disponível, forçando verificações periódicas.

3.4 Escalabilidade limitada

À medida que o número de threads aumenta, a contenção pelo mutex também aumenta, resultando em maior overhead de sincronização e possivelmente maior contention dos recursos do sistema.

4 Solução com Semáforos

Os semáforos permitem uma sincronização mais eficiente, removendo a necessidade de busy waiting:

```
sem_t empty_slots;  // Contador de espaços vazios
sem_t filled_slots; // Contador de espaços preenchidos
pthread_mutex_t buffer_mutex = PTHREAD_MUTEX_INITIALIZER;

// Inicialização
sem_init(&empty_slots, 0, TAMANHO - 1);
sem_init(&filled_slots, 0, 0);

void *produtor(void *arg) {
    int v;
    for (v = 1;; v++) {
        // Espera até haver um slot vazio
        sem_wait(&empty_slots);

        // Protege a seção crítica
        pthread_mutex_lock(&buffer_mutex);

        printf("Produzindo %d\n", v);
        dados[inserir] = v;
        inserir = (inserir + 1) % TAMANHO;

        pthread_mutex_unlock(&buffer_mutex);
```

```
23
24          // Sinaliza que há um novo item
25          sem_post(&filled_slots);
26
27          usleep(500000);
28      }
29      return NULL;
30  }
31
32  void *consumidor(void *arg) {
33      for (;;) {
34          // Espera até haver um item disponível
35          sem_wait(&filled_slots);
36
37          // Protege a seção crítica
38          pthread_mutex_lock(&buffer_mutex);
39
40          printf("%zu: Consumindo %d\n", (size_t)arg, dados[
    remover]);
41          remover = (remover + 1) % TAMANHO;
42
43          pthread_mutex_unlock(&buffer_mutex);
44
45          // Sinaliza que há um novo slot vazio
46          sem_post(&empty_slots);
47      }
48      return NULL;
49  }
```

Esta solução usa dois semáforos:

- **empty_slots**: Conta o número de slots vazios no buffer.
- **filled_slots**: Conta o número de slots preenchidos no buffer.

Vantagens desta solução:

- **Sem busy waiting**: As threads são bloqueadas pelo sistema operacional até que a operação `sem_wait()` possa prosseguir.
- **Sinalização direta**: Existe uma sinalização explícita quando novos itens são produzidos ou slots são liberados.
- **Melhor eficiência**: As threads são bloqueadas quando não podem progredir, liberando o processador para outras tarefas.

- **Melhor escalabilidade**: Reduz a contenção pelo mutex, pois as threads só acessam o mutex quando têm garantia de que podem progredir.

5 Solução com Variáveis de Condição

As variáveis de condição oferecem outra abordagem eficiente para sincronização, permitindo que as threads aguardem condições específicas:

```
pthread_mutex_t buffer_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
volatile int count = 0;  // Número de itens no buffer

void *produtor(void *arg) {
    int v;
    for (v = 1;; v++) {
        pthread_mutex_lock(&buffer_mutex);

        // Espera enquanto o buffer estiver cheio
        while (count == TAMANHO - 1)
            pthread_cond_wait(&not_full, &buffer_mutex);

        printf("Produzindo %d\n", v);
        dados[inserir] = v;
        inserir = (inserir + 1) % TAMANHO;
        count++;

        // Sinaliza que o buffer não está mais vazio
        pthread_cond_signal(&not_empty);

        pthread_mutex_unlock(&buffer_mutex);

        usleep(500000);
    }
    return NULL;
}

void *consumidor(void *arg) {
    for (;;) {
        pthread_mutex_lock(&buffer_mutex);

        // Espera enquanto o buffer estiver vazio
        while (count == 0)
            pthread_cond_wait(&not_empty, &buffer_mutex);
```

```
37
38          printf("%zu: Consumindo %d\n", (size_t)arg, dados[
    remover]);
39          remover = (remover + 1) % TAMANHO;
40          count--;
41
42          // Sinaliza que o buffer não está mais cheio
43          pthread_cond_signal(&not_full);
44
45          pthread_mutex_unlock(&buffer_mutex);
46      }
47      return NULL;
48  }
```

Esta solução utiliza:

- Um mutex para proteger o acesso às variáveis compartilhadas.
- Duas variáveis de condição:
  - **not_empty**: Sinalizada quando itens são adicionados ao buffer.
  - **not_full**: Sinalizada quando itens são removidos do buffer.
- Uma variável `count` que mantém o número atual de itens no buffer.

Vantagens desta solução:

- **Eficiência**: Como os semáforos, não há busy waiting. As threads são bloqueadas eficientemente.
- **Flexibilidade**: As variáveis de condição permitem verificar predicados mais complexos além de simples contadores.
- **Atomicidade**: A função `pthread_cond_wait()` libera o mutex automaticamente enquanto a thread está bloqueada e o readquire quando ela acorda, garantindo atomicidade.
- **Sinalização precisa**: As threads são sinalizadas exatamente quando as condições mudam, evitando verificações desnecessárias.

6 Conclusão

O problema do produtor-consumidor ilustra desafios fundamentais de sincronização em sistemas concorrentes. Comparando as soluções:

- A solução original tem problemas de corrida críticos e é ineficiente devido ao busy waiting.
- A solução com mutex resolve os problemas de corrida, mas mantém o problema de eficiência com um tipo modificado de busy waiting.
- As soluções com semáforos e variáveis de condição resolvem tanto os problemas de corrida quanto de eficiência, evitando o busy waiting.

Tanto semáforos quanto variáveis de condição são adequados para este problema, com variáveis de condição oferecendo maior flexibilidade para condições complexas. A escolha entre eles geralmente depende dos requisitos específicos do sistema e das preferências de design.