Anotações Técnicas: Implementação e Otimização de Mecanismos de Sincronização

Sistemas Operacionais

27 de junho de 2025

1 Introdução

Este documento apresenta anotações técnicas detalhadas sobre as implementações do problema do produtor-consumidor utilizando diferentes mecanismos de sincronização. As anotações explicam os aspectos críticos de cada implementação, destacando as operações de baixo nível e seus impactos no desempenho.

2 Problema Original (Sem Sincronização Adequada)

```c
volatile int dados[TAMANHO];
volatile size_t inserir = 0;
volatile size_t remover = 0;

void *produtor(void *arg) {
    int v;
    for (v = 1;; v++) {
        /* PROBLEMA: Busy waiting sem sincronização
         * - O uso de 'volatile' não garante atomicidade
         * - Múltiplas threads podem verificar a condição simultaneamente
         * - Cache ping-pong ocorre entre os núcleos da CPU
         */
        while (((inserir + 1) % TAMANHO) == remover);

        /* SEÇÃO CRÍTICA SEM PROTEÇÃO
         * - Race condition: outra thread pode modificar 'remover' entre
         *   a verificação acima e a operação de escrita
         * - Não há garantia de visibilidade entre núcleos sem barreira de memória
         */
        printf("Produzindo %d\n", v);
        dados[inserir] = v;  // Escrita no buffer compartilhado
        inserir = (inserir + 1) % TAMANHO;  // Atualização do índice

        usleep(500000);  // Sleep para simular processamento
    }
    return NULL;
}

void *consumidor(void *arg) {
```

```
    for (;;) {
        /* PROBLEMA: Busy waiting sem sincronização
         * - Mesmos problemas do produtor
         * - Gasto desnecessário de CPU em verificações contínuas
         */
        while (inserir == remover);

        /* SEÇÃO CRÍTICA SEM PROTEÇÃO
         * - Múltiplos consumidores podem ler o mesmo valor
         * - Race condition na atualização de 'remover'
         */
        printf("%zu: Consumindo %d\n", (size_t)arg, dados[remover]);
        remover = (remover + 1) % TAMANHO;  // Atualização do índice
    }
    return NULL;
}
```

Listing 1: Implementação Original com Problemas

3 Solução com Mutex

```
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) {
            /* OPERAÇÃO DE BAIXO NÍVEL: pthread_mutex_lock()
             * 1. Executa instrução atômica (ex: LOCK CMPXCHG em x86)
             * 2. Se lock já estiver ocupado:
             *    - Em implementações modernas: tenta spin por um curto período
             *    - Depois faz syscall para bloquear a thread (transição para kernel)
             *    - Thread entra na fila do mutex
             * 3. Impacto no hardware:
             *    - Barreira de memória completa (fence)
             *    - Invalidação de cache para outros núcleos
             */
            pthread_mutex_lock(&buffer_mutex);

            /* VERIFICAÇÃO PROTEGIDA
             * - Thread tem acesso exclusivo ao estado do buffer
             * - Estado é consistente e visível para esta thread
             */
            if (((inserir + 1) % TAMANHO) != remover) {
                can_insert = 1;

                /* SEÇÃO CRÍTICA PROTEGIDA
                 * - Garantia de exclusividade durante as operações
                 */
                printf("Produzindo %d\n", v);
                dados[inserir] = v;
                inserir = (inserir + 1) % TAMANHO;
```

```
            }

            /* OPERAÇÃO DE BAIXO NÍVEL: pthread_mutex_unlock()
             * 1. Executa instrução atômica para liberar o lock
             * 2. Se houver threads esperando:
             *    - Sinaliza para o kernel acordar uma thread (syscall)
             * 3. Impacto no hardware:
             *    - Barreira de memória (garante visibilidade das alterações)
             *    - Novas invalidações de cache
             */
            pthread_mutex_unlock(&buffer_mutex);

            if (!can_insert) {
                /* ESPERA COM REDUÇÃO DE CONTENÇÃO
                 * - Não é eficiente como semáforos/condições
                 * - Reduz impacto do busy waiting, mas não elimina
                 */
                usleep(10000);  // Sleep por 10ms
            }
        }

        usleep(500000);  // Sleep para simular processamento
    }
    return NULL;
}

void *consumidor(void *arg) {
    /* Implementação similar ao produtor com mesmos padrões
     * de aquisição/liberação de mutex e verificação protegida
     */
    // ... código omitido para brevidade ...
}
```

Listing 2: Implementação com Mutex

4 Solução com Semáforos

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

/* INICIALIZAÇÃO
 * - empty_slots: inicia com TAMANHO-1 (slots vazios disponíveis)
 * - filled_slots: inicia com 0 (nenhum slot preenchido)
 */
sem_init(&empty_slots, 0, TAMANHO - 1);
sem_init(&filled_slots, 0, 0);

void *produtor(void *arg) {
    int v;
    for (v = 1;; v++) {
        /* OPERAÇÃO DE BAIXO NÍVEL: sem_wait(&empty_slots)
         * 1. Tenta decrementar o contador do semáforo (operação atômica)
         * 2. Se contador >= 1: decrementa e prossegue
```

```
         * 3. Se contador = 0:
         *    a. Thread é bloqueada pelo kernel (syscall)
         *    b. Thread é colocada na fila do semáforo
         *    c. Thread só será acordada quando semáforo > 0
         * 4. Evita completamente busy waiting
         * 5. Mais eficiente que mutex para sinalização
         */
        sem_wait(&empty_slots);  // Espera por slot vazio

        /* MUTEX AINDA NECESSÁRIO
         * - Semáforo controla disponibilidade de recursos
         * - Mutex protege acesso concorrente às variáveis
         */
        pthread_mutex_lock(&buffer_mutex);

        /* SEÇÃO CRÍTICA
         * - Garantida para executar apenas quando há espaço
         * - Não precisa verificar condição novamente
         */
        printf("Produzindo %d\n", v);
        dados[inserir] = v;
        inserir = (inserir + 1) % TAMANHO;

        pthread_mutex_unlock(&buffer_mutex);

        /* OPERAÇÃO DE BAIXO NÍVEL: sem_post(&filled_slots)
         * 1. Incrementa contador do semáforo atomicamente
         * 2. Se threads estiverem esperando neste semáforo:
         *    - Kernel acorda uma das threads (ou mais, dependendo da impl.)
         *    - Transferência mais eficiente que polling
         * 3. Sinalização direta - o consumidor saberá exatamente
         *    quando há dados disponíveis
         */
        sem_post(&filled_slots);  // Sinaliza que um novo item está disponível

        usleep(500000);  // Sleep para simular processamento
    }
    return NULL;
}

void *consumidor(void *arg) {
    /* Padrão complementar ao produtor:
     * - Espera por itens (sem_wait(&filled_slots))
     * - Processa item com mutex protegendo a seção crítica
     * - Sinaliza que um slot ficou vazio (sem_post(&empty_slots))
     */
    // ... código omitido para brevidade ...
}
```

Listing 3: Implementação com Semáforos

5 Solução com Variáveis de Condição

```
pthread_mutex_t buffer_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```c
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
volatile int count = 0;  // Contador de itens no buffer

void *produtor(void *arg) {
    int v;
    for (v = 1;; v++) {
        /* Mutex necessário para acessar variáveis de condição
         * - As variáveis de condição sempre operam em conjunto com um mutex
         * - Mutex protege a verificação da condição
         */
        pthread_mutex_lock(&buffer_mutex);

        /* OPERAÇÃO DE BAIXO NÍVEL: pthread_cond_wait()
         * 1. Atomicamente:
         *    a. Libera o mutex
         *    b. Coloca a thread para dormir
         * 2. Quando sinalizada (e mutex disponível):
         *    a. Reacquire o mutex
         *    b. Retorna da função
         * 3. Loop while necessário para verificação de predicate
         *    (proteção contra spurious wakeups)
         * 4. Vantagens sobre semáforos:
         *    - Permite predicados mais complexos
         *    - Broadcast para múltiplas threads
         */
        while (count == TAMANHO - 1) {
            pthread_cond_wait(&not_full, &buffer_mutex);
        }

        /* SEÇÃO CRÍTICA
         * - Thread possui o mutex neste ponto
         * - Condição garantida: há espaço no buffer
         */
        printf("Produzindo %d\n", v);
        dados[inserir] = v;
        inserir = (inserir + 1) % TAMANHO;
        count++;

        /* OPERAÇÃO DE BAIXO NÍVEL: pthread_cond_signal()
         * 1. Sinaliza para uma thread esperando na condição
         * 2. Se múltiplas threads estão esperando:
         *    - Apenas uma é acordada (escolha implementação-dependente)
         *    - Para acordar todas: pthread_cond_broadcast()
         * 3. Se nenhuma thread está esperando: sinal é perdido
         * 4. O mutex NÃO é liberado pelo signal/broadcast
         */
        pthread_cond_signal(&not_empty);  // Sinaliza que o buffer não está mais vazio

        pthread_mutex_unlock(&buffer_mutex);

        usleep(500000);  // Sleep para simular processamento
    }
    return NULL;
}

```

```c
void *consumidor(void *arg) {
    /* Padrão complementar ao produtor:
     * - Adquire mutex
     * - Espera pela condição not_empty enquanto buffer vazio
     * - Processa item
     * - Sinaliza not_full
     * - Libera mutex
     */
    // ... código omitido para brevidade ...
}
```

Listing 4: Implementação com Variáveis de Condição

6 Comparação de Desempenho

Mecanismo	**Uso de CPU**	**Latência**	**Operações de baixo nível**
Mutex com busy waiting	Alto	Baixa para verificação Alta para bloqueio	Instruções atômicas frequentes Muitas invalidações de cache
Mutex com sleep	Médio	Média	Syscalls para sleep Várias tentativas de lock
Semáforos	Baixo	Otimizada	Bloqueio no kernel Sinalização direta
Variáveis de Condição	Baixo	Otimizada	Bloqueio no kernel Verificação de predicado

7 Instruções Atômicas e Barreiras de Memória

As operações de sincronização sempre usam instruções atômicas e barreiras de memória:

```
# Instrução para aquisição de mutex (pseudocódigo assembly x86)
lock_mutex:
    # LOCK prefix garante atomicidade entre processadores
    LOCK CMPXCHG [mutex_addr], 1   # Se [mutex_addr]==0, então [mutex_addr]=1
    JNZ wait_or_yield              # Se já estava bloqueado, espera ou cede CPU

    # Barreira de memória implícita em LOCK
    # Todos os loads/stores anteriores são visíveis
    # antes de qualquer load/store posterior

    # Continua execução com lock adquirido

# Instrução para liberação de mutex
unlock_mutex:
    # Barreira de memória antes de liberar o lock
    MFENCE                         # Memory Fence - todas operações completam
```

```
17
18      MOV [mutex_addr], 0           # Libera o lock
19
20      # Sinaliza outras threads (via OS) se necessário
```

Listing 5: Exemplos de Instruções x86 para Sincronização

8 Conclusão e Melhores Práticas

Baseado na análise das diferentes implementações, podemos extrair as seguintes melhores práticas:

1. Evite busy waiting em código de produção, exceto para seções críticas extremamente curtas.
2. Prefira semáforos quando o padrão de uso envolve contagem de recursos e sinalização entre threads.
3. Use variáveis de condição quando precisa verificar predicados complexos ou fazer broadcast.
4. Mantenha seções críticas o mais curtas possível para minimizar contenção.
5. Considere o alinhamento de cache (padding) em estruturas de sincronização para evitar falso compartilhamento.
6. Em sistemas embarcados ou de tempo real, esteja ciente das inversões de prioridade que podem ocorrer com mutex.
7. Considere mecanismos lock-free para casos específicos de alto desempenho.

A escolha do mecanismo de sincronização correto tem impacto significativo não apenas na correção do programa, mas também em seu desempenho, escalabilidade e eficiência energética.