

Segundo Estudo Dirigido

Programação Concorrente (ICP-361) - 2025-2 - Profa. Silvana Rossetto

¹IC/CCMN/UFRJ – 7 de outubro de 2025

Questão 1 O código abaixo implementa uma solução para o padrão **produtor/consumidor** (funções *insere* e *retira* no buffer compartilhado) com a seguinte variação do problema: a cada execução de um **consumidor**, ele deve consumir o buffer inteiro, e não apenas um único item (para isso ele deve esperar o buffer ficar completamente cheio). O produtor segue a lógica convencional, isto é, insere um item de cada vez. A aplicação poderá ter mais de um produtor e mais de um consumidor. Responda, justificando: Essa solução está correta? Ela atende a todos os requisitos do problema?

```
// Variaveis globais
sem_t mutex;      //inicia com 1 sinal
sem_t slotCheio; //inicia com 0 sinais
sem_t slotVazio; //inicia com N sinais

int Buffer[N];

void Insere (int item) {
    static int in=0;
    sem_wait(&slotVazio);
    sem_wait(&mutex);
    Buffer[in] = item;
    in++;
    if(in==N) {
        in=0;
        sem_post(&slotCheio);
    }
    sem_post(&mutex);
}

void Retira (int itens[]) {
    sem_wait(&slotCheio);
    for(int i=0; i<N; i++) {
        itens[i] = Buffer[i];
        sem_post(&slotVazio);
    }
}
```

Questão 2 O código abaixo implementa a lógica central de operação de dois tipos de threads: *foo* e *bar*. Descubra qual é a condição para elas executarem a linha //SC: usa o recurso. Justifique sua resposta.

```
//globais
int a=0, b=0; //numero de threads foo e bar usando o recurso
sem_t emA, emB; //semaforos para exclusao mutua
sem_t rec; //semaforo para sincronizacao logica
//inicializacoes que devem ser feitas na main()
//antes da criacao das threads
sem_init(&emA, 0, 1);
sem_init(&emB, 0, 1);
sem_init(&rec, 0, 1);

void *foo () {
    while(1) {
        sem_wait(&emA);
        a++;
        if(a==1) {
            sem_wait(&rec);
        }
        sem_post(&emA);
        //SC: usa o recurso
        sem_wait(&emA);
        a--;
        if(a==0) sem_post(&rec);
        sem_post(&emA);
    }
}

void *bar () {
    while(1) {
        sem_wait(&emB);
        b++;
        if(b==1) {
            sem_wait(&rec);
        }
        sem_post(&emB);
        //SC: usa o recurso
        sem_wait(&emB);
        b--;
        if(b==0) sem_post(&rec);
        sem_post(&emB);
    }
}
```

Questão 3 Considere uma **aplicação concorrente em C** com N threads que faz acesso de leitura e escrita em uma base de dados compartilhada pelas threads. O programa abaixo implementa as funções que as threads deverão executar ANTES e DEPOIS de fazerem as operações de leitura e escrita nessa base de dados, de forma a garantir os seguintes requisitos: (i) mais de uma operação de leitura pode ocorrer ao mesmo tempo; (ii) apenas uma operação de escrita pode ocorrer de cada vez; (iii) nenhuma operação de leitura pode ocorrer enquanto uma operação de escrita está sendo realizada; (iv) as operações de escrita devem ser priorizadas (novas operações de leitura só podem ser iniciadas quando não houver mais operações de escrita em espera). Avalie o programa apresentado e responda as questões abaixo justificando suas respostas: **(a)** Os requisitos (i) e (ii) são atendidos no programa? **(b)** O requisito (iv) é atendido no programa? **(c)** Há possibilidade de ocorrência de *starvation* (inanição) em alguma execução deste programa?

Inanização ocorre quando um fluxo de execução tem sua execução postergada por um longo período de tempo porque outros fluxos têm maior prioridade ou conseguem alocar os recursos primeiro. Não é um problema crítico, que gera erros lógicos, por isso pode existir nos programas concorrentes. Entretanto, se essa espera longa causar impacto negativo ou não for desejada, então o algoritmo do programa deve ser revisto.

```
sem_t em_e, em_l, escr, prioridade; //semaforos iniciados com 1 sinal
int e=0, l=0;

void AntesLeitura() {
    sem_wait(&prioridade);
    sem_wait(&em_l);
    l++;
    if(l==1) sem_wait(&escr);
    sem_post(&em_l);
    sem_post(&prioridade);
}

void DepoisLeitura() {
    sem_wait(&em_l);
    l--;
    if(l==0) sem_post(&escr);
    sem_post(&em_l);
}

void AntesEscrita() {
    sem_wait(&em_e);
    e++;
    if(e==1) sem_wait(&prioridade);
    sem_post(&em_e);
    sem_wait(&escr);
}

void DepoisEscrita() {
    sem_post(&escr);
    sem_wait(&em_e);
    e--;
    if(e==0) sem_post(&prioridade);
    sem_post(&em_e);
}
```

Questão 4 O código abaixo¹ propõe uma implementação de *variáveis de condição* e suas operações básicas wait, signal e broadcast fazendo uso de semáforos. A operação wait deve liberar o lock atualmente detido pela thread e bloquear essa thread. A operação signal deve verificar se há alguma thread bloqueada na variável de condição e desbloqueá-la. A operação broadcast deve verificar se há threads bloqueadas na variável de condição e desbloquear todas elas. Considerando o que foi exposto, examine esse algoritmo e **responda justificando:** **(a)** Qual é a finalidade dos semáforos *s*, *x* e *h* dentro do código? Esses semáforos foram inicializados corretamente? **(b)** Essa implementação está correta? Ela garante que a semântica das operações *wait*, *signal* e *broadcast* está sendo atendida plenamente? **(c)** Existe a possibilidade de acúmulo indevido de sinais nos semáforos *s*, *x* e *h*?

```
//variaveis internas
sem_t s, x, h;    int aux = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

//inicializacoes feitas na funcao principal
sem_init(&s,0,0); sem_init(&x,0,1); sem_init(&h,0,0);

void wait() {
    //pre-condicao: a thread corrente detem o lock de 'm'
    sem_wait(&x);
    aux++;
    sem_post(&x)
    pthread_mutex_unlock(&m);
    sem_wait(&h);
    sem_post(&s);
    pthread_mutex_lock(&m);
}

void signal() {
    sem_wait(&x);
    if (aux > 0) {
        aux--;
        sem_post(&h);
        sem_wait(&s);
    }
    sem_post(&x)
}

void broadcast() {
    sem_wait(&x);
    for (int i = 0; i < aux; i++)
        sem_post(&h);
    while (aux > 0) {
        aux--;
        sem_wait(&s);
    }
    sem_post(&x);
}
```

¹ Adaptado de Birrell, Andrew D. "Implementing condition variables with semaphores." Computer Systems. Springer, New York, NY, 2004. 29-37.