

UNIVERSIDADE FEDERAL DE VIÇOSA

INF310 – PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

Lista de Exercícios 4

SESSÃO 1 - QUESTÕES TEÓRICAS

1. Sobre as operações fundamentais de semáforos e variáveis de condição:

- a) Explique por que `wait()` em semáforos decrementa atômicamente e bloqueia, enquanto `wait()` em `condition_variable` precisa liberar explicitamente o mutex antes de bloquear.
- b) Compare a fila de espera em semáforos com a fila em `condition_variables`. Como cada uma mantém a ordem e por que `condition_variables` podem sofrer `spurious wakeups`?
- c) Descreva o que acontece internamente quando `post()` é chamado em um semáforo com múltiplas threads bloqueadas versus quando `notify_one()` é chamado.

2. Sobre implementação de primitivas de sincronização:

- a) Por que implementações modernas de semáforos usam `futex` (fast userspace mutex) no Linux ao invés de sempre fazer `syscalls`? Explique o caminho rápido e o caminho lento.
- b) Explique por que `condition_variable.wait()` precisa ser chamada dentro de um loop verificando um predicado. Relacione isso com `spurious wakeups` e sinais perdidos.
- c) Qual a relação entre `memory ordering` (acquire/release semantics) e a implementação correta de semáforos em arquiteturas com memória fracamente ordenada?

3. Sobre diferenças arquiteturais entre semáforos e monitores:

- a) Monitores em Java usam o modelo "signal-and-continue" (Mesa semantics) enquanto alguns sistemas usam "signal-and-wait" (Hoare semantics). Explique as diferenças e implicações de cada modelo.
- b) Por que um monitor pode ser implementado usando semáforos, mas o contrário não é verdadeiro sem estruturas adicionais?
- c) Compare o overhead de troca de contexto ao usar semáforos versus monitores em cenários de alta contenção.

4. Sobre problemas de sincronização e escolha de primitivas:

- a) Explique por que o problema do barbeiro dorminhoco é mais elegantemente resolvido com semáforos do que com monitores, detalhando as dificuldades com cada abordagem.
- b) Descreva situações onde `notify_all()` é obrigatório ao invés de `notify_one()` e explique o problema de "thundering herd" que isso pode causar.

- c) Como semáforos de leitura-escrita (read-write locks) diferem de semáforos contadores comuns em termos de semântica e implementação?

5. Sobre garantias de atomicidade e corretude:

- a) Por que a operação de verificar um contador e decrementá-lo em um semáforo deve ser indivisível? O que aconteceria se fossem duas operações separadas?
 - b) Explique o problema de "lost wakeup" em `condition_variables` e como o padrão de uso (predicado em `loop + mutex`) previne isso.
 - c) Como a ausência de `ownership` em semáforos binários pode levar a bugs sutis que não ocorrem com `mutexes`? Dê exemplos concretos.
-

SESSÃO 2 - QUESTÕES DE MÚLTIPLA ESCOLHA

1. Um semáforo `S` é inicializado com valor 5. As operações `wait(S)`, `wait(S)`, `post(S)`, `wait(S)`, `wait(S)`, `wait(S)`, `wait(S)` são executadas nesta ordem. Qual o estado final?

- a) `S = 0`, nenhuma thread bloqueada
- b) `S = 0`, 1 thread bloqueada
- c) `S = -1`, 1 thread bloqueada
- d) `S = 1`, nenhuma thread bloqueada

2. Em um monitor, se uma thread `T1` está executando um método e chama `wait()` em uma `condition_variable`:

- a) `T1` mantém o lock do monitor e bloqueia
- b) `T1` libera o lock, bloqueia, e readquire o lock ao acordar
- c) `T1` libera o lock permanentemente
- d) Causa deadlock

3. Qual a principal razão para `condition_variable.wait()` ser sempre usada dentro de um loop?

- a) Melhor performance
- b) Spurious wakeups e race conditions
- c) Exigência do compilador
- d) Compatibilidade com versões antigas

4. Um sistema tem 3 recursos idênticos. Para controlar o acesso, o melhor é:

- a) 3 `mutexes` separados
- b) 1 semáforo contador inicializado com 3
- c) 1 semáforo binário

- d) 3 condition_variables

5. Qual afirmação sobre semáforos binários vs mutexes é INCORRETA?

- a) Semáforos binários não têm conceito de ownership
- b) Qualquer thread pode fazer post() em um semáforo binário
- c) Mutexes podem detectar unlock por thread diferente do lock
- d) Semáforos binários são sempre mais eficientes que mutexes

6. No problema produtor-consumidor com buffer limitado, quantos semáforos são minimamente necessários?

- a) 1 (apenas exclusão mútua)
- b) 2 (espaços vazios e itens cheios)
- c) 3 (espaços vazios, itens cheios, exclusão mútua)
- d) 4 (um para cada operação)

7. Quando notify_one() é chamado em uma condition_variable sem threads aguardando:

- a) A próxima thread que chamar wait() não bloqueia
- b) O sinal é perdido
- c) Uma exceção é lançada
- d) O sistema operacional enfileira o sinal

8. Em que situação um semáforo inicializado com 0 é útil?

- a) Exclusão mútua
- b) Barreira de sincronização entre threads
- c) Pool de recursos
- d) Nunca é útil

9. Considere: `sem_wait(s); x++; sem_post(s);` onde s foi inicializado com 1. Isso implementa:

- a) Incremento atômico de x
- b) Seção crítica protegendo x
- c) Sincronização de eventos
- d) Deadlock garantido

10. Um monitor implementa implicitamente:

- a) Apenas exclusão mútua nos métodos
- b) Exclusão mútua e variáveis de condição
- c) Prioridades entre threads

- d) Prevenção de deadlock

11. Em arquiteturas multicore, por que desabilitar interrupções NÃO é suficiente para implementar seções críticas?

- a) Interrupções não podem ser desabilitadas
- b) Outros cores continuam executando
- c) É muito lento
- d) Causa deadlock

12. O problema de inversão de prioridade ocorre quando:

- a) Thread de alta prioridade bloqueia esperando recurso de thread de baixa prioridade
- b) Todas as threads têm mesma prioridade
- c) Semáforos são usados incorretamente
- d) Há muitas threads executando

13. Qual operação NÃO precisa ser atômica em um semáforo?

- a) Verificar se contador > 0
- b) Decrementar contador
- c) Enfileirar thread bloqueada
- d) Nenhuma das anteriores (todas precisam ser atômicas juntas)

14. Para implementar uma barreira onde N threads devem sincronizar, são necessários:

- a) 1 semáforo
- b) 2 semáforos
- c) N semáforos
- d) N-1 semáforos

15. Qual NÃO é uma vantagem de monitores sobre semáforos?

- a) Encapsulamento de dados e sincronização
- b) Menor propensão a erros de ordem
- c) Melhor performance em todos os casos
- d) Código mais estruturado

16. Em C++, `std::lock_guard` NÃO pode ser usado com `condition_variable` porque:

- a) É muito lento
- b) Não permite unlock/relock durante wait()

- c) Não é thread-safe
- d) Foi descontinuado

17. Um semáforo pode entrar em estado negativo?

- a) Não, nunca
- b) Sim, indica threads bloqueadas
- c) Apenas em implementações antigas
- d) Depende da implementação, mas conceitualmente pode

18. No problema leitores-escritores, para permitir múltiplos leitores simultâneos:

- a) Cada leitor precisa de um mutex próprio
- b) Usa-se contador de leitores + semáforo/mutex
- c) Não é possível com primitivas padrão
- d) Semáforo binário é suficiente

19. Qual problema clássico demonstra a necessidade de ordenação de aquisição de recursos?

- a) Produtor-consumidor
- b) Leitores-escritores
- c) Jantar dos filósofos
- d) Barbeiro dorminhoco

20. Memory barriers/fences são necessários em implementações de semáforos para:

- a) Melhorar performance
- b) Garantir visibilidade de escritas entre cores
- c) Prevenir interrupções
- d) Compatibilidade com hardware antigo

SESSÃO 3 - CONVERSÃO DE ESTRUTURAS

1. Converta o código abaixo para usar SEMÁFOROS ao invés de variáveis globais:

```
cpp
```

```
int contador = 0;
bool disponivel = true;

void incrementa() {
    while(true) {
        while(!disponivel); // busy wait
        disponivel = false;
        contador++;
        disponivel = true;
    }
}

void decrementa() {
    while(true) {
        while(!disponivel); // busy wait
        disponivel = false;
        contador--;
        disponivel = true;
    }
}
```

2. Converta o código abaixo para usar um MONITOR:

cpp

```

mutex m;
int leitores = 0;

void inicia_leitura() {
    m.lock();
    leitores++;
    m.unlock();
}

void termina_leitura() {
    m.lock();
    leitores--;
    m.unlock();
}

void escreve() {
    m.lock();
    while(leitores > 0) {
        m.unlock();
        // espera ocupada
        m.lock();
    }
    // escreve
    m.unlock();
}

```

3. Converta a sincronização abaixo para usar MONITOR ao invés de busy-waiting:

```

cpp

bool recurso_livre = true;
int fila_espera = 0;

void usa_recurso() {
    while(!recurso_livre) {
        fila_espera++;
        // busy wait
        fila_espera--;
    }
    recurso_livre = false;
    // usa recurso
    recurso_livre = true;
}

```

4. Converta para usar SEMÁFOROS:

cpp

```
int vagas = 5;

void estaciona() {
    while(vagas <= 0); // espera vaga
    vagas--;
    // estaciona
}

void sai() {
    vagas++;
}
```

SESSÃO 4 - IMPLEMENTAÇÃO A PARTIR DE CENÁRIOS

Questões com MONITORES

1. Um sistema de chat tem salas de conversa com capacidade máxima de 10 usuários. Quando um usuário tenta entrar em uma sala cheia, ele aguarda até que alguém saia. Implemente um monitor que controla a entrada e saída de usuários de uma sala. O monitor deve ter os métodos `entrar()` e `sair()`. Múltiplas threads (usuários) tentarão entrar e sair concorrentemente.
2. Uma clínica veterinária atende cães e gatos, mas por política da clínica, cães e gatos não podem estar na sala de espera ao mesmo tempo. Se há cães esperando, novos gatos devem aguardar até que todos os cães sejam atendidos e vice-versa. Implemente um monitor com métodos `entra_cao()`, `sai_cao()`, `entra_gato()`, `sai_gato()`.
3. Um restaurante self-service tem 4 balanças para pesar pratos. Clientes pegam uma balança, pesam, e liberam. Se todas estiverem ocupadas, o cliente aguarda. Adicionalmente, existe um funcionário que periodicamente precisa calibrar todas as 4 balanças simultaneamente, e para isso, todas devem estar livres. Implemente um monitor que sincroniza clientes e o funcionário de calibração.

Questões com SEMÁFOROS

4. Um jogo online precisa formar times de exatamente 5 jogadores antes de iniciar uma partida. Jogadores chegam aleatoriamente e aguardam. Quando o 5º jogador chega, todos os 5 são liberados para jogar juntos e o processo recomeça para formar o próximo time. Implemente usando semáforos a sincronização entre a thread principal (que detecta times completos) e as threads dos jogadores.
5. Uma ponte de mão única suporta no máximo 3 veículos por vez, mas todos devem estar indo na mesma direção. Veículos chegam de ambos os lados (Norte e Sul). Quando a ponte está vazia, o próximo veículo (de qualquer lado) define a direção. Quando há veículos na ponte indo em uma direção, outros da mesma direção podem entrar (até 3 total), mas os da direção oposta devem aguardar. Implemente usando semáforos a sincronização para threads representando veículos do Norte e do Sul.

6. Uma empresa tem 2 impressoras: uma laser (rápida) e uma jato de tinta (lenta). Existem 3 tipos de documentos: urgentes (só podem usar laser), normais (preferem laser, mas aceitam jato), e rascunhos (aceitam qualquer uma). Implemente com semáforos a sincronização onde threads representando documentos competem pelas impressoras respeitando essas preferências. Documente claramente os semáforos necessários e seus valores iniciais.

GABARITO - SESSÃO 2

1. **b)** $S = 0$, 1 thread bloqueada. Operações: $5 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$ (7ª wait bloqueia)
2. **b)** T1 libera o lock, bloqueia, e readquire ao acordar. Comportamento padrão de condition_variable
3. **b)** Spurious wakeups e race conditions. Wait pode acordar sem notify e condição pode mudar entre notify e relock
4. **b)** 1 semáforo contador inicializado com 3. Controla quantidade de recursos disponíveis
5. **d)** Semáforos binários NÃO são sempre mais eficientes. Depende do caso de uso e implementação
6. **c)** 3 semáforos: empty (espaços livres), full (itens disponíveis), mutex (exclusão mútua no buffer)
7. **b)** O sinal é perdido. Condition variables não armazenam sinais
8. **b)** Barreira/sincronização de eventos. Thread aguarda até outra fazer post
9. **b)** Seção crítica protegendo x. Garante acesso exclusivo ao incremento
10. **b)** Exclusão mútua e variáveis de condição. Monitor encapsula ambos
11. **b)** Outras cores continuam executando. Interrupções são por core, não globais
12. **a)** Thread alta prioridade bloqueada esperando recurso de baixa prioridade que não executa
13. **d)** Nenhuma. Todas as operações de test-and-set devem ser atômicas juntas
14. **b)** 2 semáforos. Um para bloquear até N threads chegarem, outro para liberá-las
15. **c)** Performance não é sempre melhor. Depende do padrão de contenção e overhead
16. **b)** Não permite unlock/relock durante wait(). Unique_lock tem lock/unlock explícitos
17. **d)** Depende da implementação, mas conceitualmente pode representar threads bloqueadas como valores negativos
18. **b)** Contador de leitores + semáforo/mutex. Múltiplos leitores incrementam contador, escritor aguarda contador = 0
19. **c)** Jantar dos filósofos. Deadlock evitado com ordenação de garfos
20. **b)** Garantir visibilidade de escritas entre cores. Memory model exige sincronização explícita