Teoria

① Explique o conceito de deadlock e descreva um cenário onde este fenómeno poderá ocorrer.

Teste 2016 e 2022

Deadlock consiste numa situação em que 2 ou mais processos ficam bloqueados indefinidamente quando tentam competir por recursos partilhados, ou seja, 1 desses processos tem um recurso e os outros processos esperam por obtê-lo.

Um exemplo prático é: Thread A adquire o recurso X e a Thread B adquire o recurso Y. De seguida A tenta adquirir o recurso Y e B o recurso X e tal não é possível se as Threads A e B libertarem-se de posse dos respetivos recursos.

② Compare brevemente monitores modernos com variantes clássicas. Explique o que complica a implementação de uma barreira reutilizável usando monitores modernos.

Teste 2022

Os monitores modernos permitem uma sincronização de threads de forma mais conveniente/eficiente, são mais seguros e menos propensos a erros, evitando assim deadlocks e race conditions.

A sincronização que falei resulta em, como o processo que faz signal, depois pode correr o processo acordado ou um que queira entrar, e como tal pode alterar o estado do monitor então fazemos sempre while(!predicado()){
wait(cond);
}

No caso dos monitores clássicos, pode ocorrer com mais facilidade deadlocks e race conditions pois manter a concorrência é mais complexo.

Se um processo estiver bloqueado num wait, a seguir a um signal como o processo bloqueado, mais tarde prossegue quem fez signal e aí podem entrar mais processos no monitor. A implementação de monitores clássicos o de signal fica:

```
if (!predicado()) {
    wait();
}
```

A implementação de um Barreira reutilizável tem complicações pois é necessário garantir que todas as threads sejam sincronizadas corretamente e garantir que não há perda de recursos para a mesma volta ou seja usada, caso contrário não é vantajoso.

(3) Suponha que tem várias threads, cada uma a efetuar operações que envolvam vários recursos partilhados. Explique que problemas podem surgir e descreva uma técnica de controlo de concorrência que os resolva. EE-2022

Podem surgir race conditions, imaginem um contador que é incrementado ou que várias threads acedem a um contador, o resultado final do contador vai ser diferente, essencial pode haver sincronia nos acessos ao contador.

EE - 2022

Teste 2016

(4) Diga o que entende por Spurious wakeup, e explique as consequências na metodologia de programação com monitores. Dê um exemplo de um problema que seria resolvido trivialmente caso o fenómeno não existisse.

Spurious wakeup consiste numa thread que é acordada de sua espera (wait) de forma espontânea, sem a mesma ser notificada/sinalizada.

Imaginemos que várias threads estão numa fila à espera (usando wait) por um recurso partilhado. Cada thread aguarda que seja notificada que o recurso que quer está disponível, então se não existisse este fenómeno, as threads só seriam acordadas quando realmente o recurso estivesse livre para uso, pelo que o programa correria de forma correta.

(5) Descreva a utilidade do Selective receive em Erlang.

Teste 2022

O Selective receive em Erlang permite a escolha entre várias mensagens da sua mailbox, selecionando a que corresponde a um padrão específico.

Podendo assim lidar com comunicação assíncrona e concorrente entre processos.

(6) Explique o conceito de starvation e descreva um cenário que seja propício à ocorrência deste fenómeno

Teste 2019

Recurso

Starvation consiste num fenómeno que ocorre quando uma thread tenta avançar (para a secção crítica), ou ser executada e nunca chega a ser executada, pois outras threads adquirem o lock e passam-lhe à frente.

Um exemplo propício a essa ocorrência é quando várias threads estão a utilizar um recurso compartilhado, no entanto, a ordem de prioridades é maior para algumas. Neste caso, estas threads podem monopolizar o acesso aos recursos e por isso as threads com baixa prioridade estão num estado de starvation. (Resumindo ocorre em contenção por recursos).

(7) É comum encontrar generalizações de semáforos com as operações como acquire(n) e release(n).

Explique a semântica destas operações e compare-as com as operações clássicas.

Recurso 2022

As operações acquire(n) e release(n) são operações que tentam adquirir/libertar, respetivamente, n recursos do semáforo (incrementando/decrementando o valor do semáforo).

Comparativamente a acquire() e release() que adquirem/libertam 1 vez e bloqueia (caso acquire).

Em sem, acquire(n): Consegue adquirir os recursos se estiverem juntos, caso seja inferior ~~a~~ ao nº de recursos fica bloqueado. Com Consegue adquirir ou libertar vários recursos em simultâneo, é mais eficiente e evita deadlocks.

(8) Explique porque motivos o wait() numa variável de condição de um monitor em linguagens atuais deve ser efetuado dentro de um ciclo que (re)teste um predicado.

É essencial usar wait() em uma variável de condição de um monitor dentro de um ciclo pois, previne os Spurious Wakeups que podem acontecer e dessa forma é testado a condição para garantir que a thread não prossiga prematuramente.

Previne ainda possíveis corridas, pois uma outra thread pode alterar a condição entre o momento em que a thread foi acordada e o momento em que retoma a execução. Desta forma. Com um ciclo a thread só é liberada quando a condição desejada for verdadeira.

Recurso 2022

(9) Que solução adoptaria para evitar Starvation?

Usaria monitores, pois vários processos a tentar chegar a uma parte em lock, entra um a espera, quando fica livre.

O processo ao sair faz signal para acordar o processo seguinte que está à espera e eventualmente chegar a vez do processo que queremos que entre.

(10) Explique a utilidade de poder ter mais do que uma variável de condição num monitor e porque tal não é equivalente a usar vários monitores, cada um com uma variável de condição.

Ter mais do que um v.c ofereça maior flexibilidade no controlo de sincronização entre threads, evitando deadlocks e starvation pois garante que as threads que esperam por diferentes condições possam ser notificadas independentemente das outras. Ao usar mais do que um v.c num só monitor decidimos que as threads ficam em espera e com vários monitores com 1 v.c isso não acontece. Além do que consumimos mais recursos do sistema, memória e tempo de processamento.

Teste 2019

EE-2016

(11) Compare as abordagens usadas para construir uma abstração para uso concorrente por threads clientes em Java e processos clientes em Erlang.

A principal diferença destas abordagens é a partilha de memória.

Em Erlang tal não acontece, cada processo é independente e tem a sua própria informação. A comunicação é feita por mensagens, mensagens essas que são decifradas/resolvidas com o pattern matching de quem a recebe. Essas mensagens ficam em fila de espera e "resolvem-se" por ordem de chegada.

Já em Java, é preciso implementar um sistema de locks de forma a evitar erros de leitura e escrita em momentos que não são os devidos/pretendidos.

Teste 2019
Recurso 2016

Teste 2016

(12) É comum ouvir "como esta thread apenas vai ler as variáveis, não necessito de usar um lock".
Comente esta afirmação.

É perigoso afirmar isto, pois em ambientes multithread, acessos concorrentes a variáveis podem levar a problemas com race conditions e comportamentos inesperados. Mesmo que a thread esteja apenas a ler uma variável, se uma outra thread nesse mesmo instante modificar a variável, então a primeira thread estará a ler um valor incorreto da variável.

Portanto, é sempre importante dar locks para evitar problemas como estes e ler/escrever apenas quando é possível e de forma correta.

Apenas não é necessário locks se estiver a ser lido um objeto/variável imutável (final int x = 30)

↳ será sempre 30

(13) Exclusão mútua e ordem de execução.

Descreva e dê um exemplo de cada um deles.

Exclusão mútua é quando um processo está a executar código sobre um objeto e o tem em lock, neste caso mais ninguém pode fazer nada com esse objeto.

Ordem de execução, é quando vários processos a tentar aceder a um mesmo pedaço de código e entre o primeiro processo que fiza lock.

Exemplo de exclusão mútua: Imaginemos o guichê dos bancos e uma thread faz lock do banco para efetuar uma transferência e mais nenhuma operação avança em nenhuma conta pois o banco está "bloqueado".

Exemplo de ordem de execução é quando várias threads tentam aceder ao mesmo bocado de código uma thread faz o primeiro lock e pode ocorrer starvation pois não há garantias de que uma dada thread entrará enquanto outras estiverem a concorrer para o mesmo bocado de código.

EE-2016

Época 2024

14 Explique porque a invocação concorrente de operações que envolvem vários objetos pode causar problemas, mesmo que cada objeto se proteja individualmente com código de controlo de concorrência interno.

Ilustre o problema com um exemplo e descreva uma técnica de controlo de concorrência que o resolva.

- A invocação concorrente de operações que envolvem vários objetos pode causar problemas mesmo que cada objeto se proteja individualmente com controlo de concorrência interno.

  Isto ocorre pois pode ocorrer deadlock.

  Exemplo:

  A thread T_1 bloqueia o objeto A e depois tenta bloquear o objeto B
  A thread T_2 bloqueia o objeto B e depois tenta bloquear o objeto A

- Se T_1 e T_2 tiverem cada um bloqueado A e B respetivamente, ambas as threads ficam à espera indefinida que o outro objeto seja desbloqueado e resulta em deadlock.

Uma técnica para controlo de concorrência é usar ordenação hierárquica dos locks. Estabelecendo uma ordem global na qual os objetos devem ser bloqueados e todas as threads seguirem essa ordem.

Ou seja se T_1 bloqueia A e depois B, T_2 só bloqueia B se A estiver disponível e desta forma previne-se o deadlock.

(15) Explique os motivos porque o await() numa Condition em Java deve ser efetuado com o Lock correspondente adquirido, nomeadamente porque o comportamento do await() foi pensado de modo a não ser o programador a ter que libertar o lock antes do await().

Os motivos pelo qual await() deve ser efetuado com o lock correspondente adquirido são:

- Liberta o lock automaticamente, ou seja, quando await() é chamado, a thread liberta o lock associado automaticamente. Desta forma evita-se deadlocks e permite que outras threads adquiram o lock e sinalizem a Condition.
- Readquirir o lock, pois após ser sinalizada (signalAll() ou signal()) a thread que estava em await precisa de readquirir o lock antes de continuar a execução.

Desta forma await() consegue-se que foi pensado para que o programador não precise de libertar o lock antes de chamar await(), reduzindo menos de erros de sincronização e complexidade.

```
Void esperar {
    l.lock();
    try {
        while(!condição) {
            cond.await();
        }
    } finally {
        l.unlock();
    }
}
```

```
Void sinalizarEspera {
    l.lock();
    try {
        cond.signalAll();
    } finally {
        l.unlock();
    }
}
```