Leonardo Vecchi Meirelles - 120111ECP002

Cap 30 - Questão 1)

As variáveis condicionais são um mecanismo de sincronização que permite que os threads esperem que uma determinada condição seja verdadeira antes de prosseguir. Eles são usados em conjunto com mutexes para proteger recursos compartilhados e garantir que apenas um thread os acesse por vez. O thread em espera libera o bloqueio no mutex e bloqueia até que outro thread sinalize que a condição foi alterada. Quando sinalizado, o thread em espera readquire o lock e verifica novamente a condição antes de prosseguir.

Cap 30 - Questão 2)

O problema produtor-consumidor é um problema clássico de sincronização. Os produtores geram dados, enquanto os consumidores processam os dados. Ambos os threads devem trabalhar juntos para realizar suas tarefas. O problema surge quando existem vários produtores e consumidores e os consumidores precisam esperar que os produtores gerem dados antes de processá-los.

O problema pode ser resolvido usando técnicas de sincronização, como locks, semáforos e variáveis de condição. Uma abordagem comum é usar um buffer compartilhado, onde produtores podem inserir dados e consumidores removê-los. O desafio é garantir que o buffer seja acessado de maneira thread-safe, o que significa que produtores e consumidores não interferem entre si.

Leonardo Vecchi Meirelles - 320116CP002

Cap 30 - Questão 3)

Na primeira tentativa de código, foi utilizada apenas uma variável de condição e faz uso de um if(). Na segunda tentativa o if() é substituído por um while(). Para apenas um produtor e um consumidor, os códigos funcionam, mas ao adicionar mais threads, problemas surgem. O problema do if() aparece com mais de um consumidor. Por exemplo, um consumidor C_1 tenta executar, mas não existem dados no buffer, logo, entra em um bloqueio (wait). Em seguida, o produtor produz dados e acorda C_1. Porém, nesse instante, C_2 executa e usa os dados do buffer. Por fim, C_1, que já passou do if(), tenta usar os dados do buffer, mas eles não existem mais. Um while() verificaria que dados não existem, deixando C_1 em sleep.

Agora, o segundo problema, o uso de apenas uma variável de condição implica que, com mais threads, produtores podem acordar produtores e consumidores acordam consumidores, causando perda de eficiência. Isso ocorre pois só existe uma variável para controlar ambos os threads, podendo acordar produtores com buffer cheios e consumidores com buffer vazio

Cap 30 - Questão 4)

O código correto utiliza while() e duas variáveis de condição, uma para produtores e outra para consumidores. É importante perceber que produtores sinalizam para consumidores e vice-versa. Isso impede que o problema dos códigos anteriores ocorra

Leonardo Vecchi Meirelles - 12011ECP002

Cap 30 - Questão 5)

Primeiramente deve-se modificar as funções put() e get() para um buffer de várias posições. O método comumente utilizado se assemelha a um buffer circular. A outra modificação é feita na verificação do produtor que, ao invés de verificar se count == 1, já que era um buffer de uma posição, deve verificar se o buffer está cheio, ou seja: count == MAX, onde MAX é o tamanho do buffer.

Cap 31 - Questão 1)

Um semáforo é uma primitiva de sincronização que permite que vários threads ou processos acessem um recurso compartilhado de maneira mutuamente exclusiva. Tem um valor inteiro e suporta duas operações fundamentais: "sem_wait()" e "sem_post". A operação sem_wait() diminui o valor do semáforo e, se ele se tornar negativo, o thread ou processo de chamada é bloqueado até outro sinalizar o semáforo com a operação sem_post(), que incrementa o valor do semáforo. As operações de semáforo são atômicas e podem ser usados para proteger recursos compartilhados de acesso simultâneo

Leonardo Vecchi Meirelles - 32011ECP002

Cap 31 - Questão 2)

Um semáforo binário é um semáforo que assume apenas dois valores: 0 e 1. Pode ser usado como um lock simples ou mutex para controlar o acesso a um recurso compartilhado, onde o valor 1 representa o recurso disponível e um valor de 0 representa que está indisponível. O semáforo binário fornece exclusão mútua para a seção crítica e pode ser utilizado para evitar race conditions e deadlocks

Cap 31 - Questão 3)

Os semáforos podem ser utilizados para impor uma determinada ordem de execução entre vários segmentos ou processos. Por exemplo, podemos usar dois semáforos para garantir que um thread conclua seu trabalho antes que outro thread seja iniciado. O primeiro thread adquirirá um semáforo (inicializado em 0) antes de iniciar seu trabalho e o liberará assim que for concluído. A segunda thread aguardará neste semáforo antes de iniciar seu trabalho. Isso garante que o segundo thread não iniciará até que o primeiro thread seja concluído. Isso pode ser estendido para mais threads usando vários semáforos, com cada thread aguardando um semáforo que é liberado pelo anterior assim que ele conclui seu trabalho.

Leonardo Vecchi Meirelles - 1201JECP002

Cap 31 - Questão 4)

Os dois problemas que podem ocorrer com a implementação apresentada são: overwrites e deadlocks. No primeiro código, não existem mutexes logo, um produtor, por exemplo, pode sofrer preempção antes de incrementar "fill" para 1, assim, quando um segundo produtor gerar dados, eles serão colocados na posição zero, sobrescrevendo os dados do primeiro produtor. No segundo código são introduzidos semáforos no local errado, podendo gerar deadlocks (semáforos funcionando como mutexes). Nesse caso apresentado, como a função "sem_wait()" não devolve a lock, se um consumidor executar primeiro, ele adquire o mutex e chama sem_wait(). Como não existem dados, ele bloqueia, mas continua em posse da lock, não permitindo que nenhum outro thread execute, já que no código apresentado, primeiro verifica-se o mutex, gerando um deadlock.

Cap 31 - Questão 5)

O próximo código soluciona o problema do deadlock reduzindo o escopo da lock, mantendo o "wait" e o "signal" dos semáforos "full" e "empty" de fora. Usando o exemplo anterior, agora, o consumidor será bloqueado pelo semáforo "empty" e não terá posse da lock e, assim, quando o produtor executar, ele terá a lock disponível e funcionará corretamente, devolvendo a lock ao final e acordando o consumidor.

Leonardo Vecchi Meirelles - 12011ECP002

Cap 32 - Questão 1)

Um bug de violação de atomicidade ocorre quando uma operação que deveria ser atômica é interrompida no meio por outro processo ou thread. Isso pode levar a um comportamento incorreto ou inesperado, como corrupção de dados ou race conditions. Violações de atomicidade podem ser causados por recursos compartilhados desprotegidos ou seções críticas. Como resultado, o sistema pode se tornar instável ou travar, e o resultado do programa pode ser imprevisível.

Cap 32 - Questão 2)

A violação da ordem de execução ocorre quando a exatidão de um programa concorrente depende da ordem na qual os threads são executados. Isso pode levar a erros quando a ordem de execução esperada não é garantida pelo programa. Por exemplo, em um problema produtor-consumidor, se a thread consumidora remover um item do buffer antes que a thread produtora o tenha adicionado, o programa pode produzir resultados incorretos. Para evitar violações da ordem de execução, os programadores devem usar mecanismos de sincronização, como semáforos e mutexes, para garantir a ordem correta de execução.

Leonardo Vecchi Meirelles - 32011ECP002

Cap 32 - Questão 3)

Um deadlock é uma situação em que dois ou mais threads estão bloqueados e esperando que o outro libere recursos ou bloqueios (locks) que eles possuem. Isso pode ocorrer quando há uma espera circular, onde cada thread está esperando um recurso montido por outra thread no ciclo. As quatro condições necessários para que um deadlock ocorra são: exclusão mútua, hold and wait, ausência de preempção e espera circular.

A exclusão mútua ocorre quando um recurso é montido exclusivamente por um thread. Hold and wait ocorre quando um thread montém um recurso enquanto espera por outro. Nenhuma preempção significa que recursos não podem ser retirados à força de um thread. Por fim, espera circular é quando há uma cadeia circular de threads esperando por recursos montidos por outros threads na cadeia.

Quando essas condições são atendidas, pode ocorrer um deadlock, fazendo com que os threads fiquem presos indefinidamente.

Leonardo Vecchi Meirelles - 12011ECP002

Cap 32 - Questão 4)

A prevenção de deadlocks pode ser feita tratando uma ou mais das condições que os causam. Para evitar a condição de exclusão mútua, os recursos podem ser compartilhados em vez de serem usados exclusivamente por um processo ou thread. Para evitar a condição de hold and wait, um processo pode adquirir todos os recursos de uma só vez, em vez de adquiri-los um por um, ou os recursos podem ser liberados e adquiridos dinamicamente conforme necessário. Para evitar a condição de não preempção, um processo pode ser forçado a liberar seus recursos se estiver esperando por muito tempo. Por fim, para evitar a condição de espera circular, os recursos podem receber números exclusivos e um processo pode ser necessário para adquirir recursos em uma ordem específica para quebrar a dependência circular.

Cap 32 - Questão 5)

A prevenção de deadlock via scheduling é uma técnica de prevenção que evita deadlocks agendando cuidadosamente as solicitações de recursos dos processos para evitar cenários de alocação de recursos inseguros. O escalonador analisa as solicitações de recursos e toma a decisão de conceder a solicitação ou aguardar a disponibilidade do recurso solicitado. Se o recurso solicitado não puder ser alocado com segurança, o thread que o solicitou será bloqueado e adicionado a uma fila de espera até que o recurso fique disponível.

Ao controlar cuidadosamente quais recursos são concedidos a quais processos, o sistema pode evitar entrar em estados de deadlock. Essa técnica requer conhecimento detalhado do gráfico de alocação de recursos e pode causar degradação de desempenho devido à espera de recursos.

Leonardo Vecchi Meirelles - 12011ECP002

Cap 32 - Questão 6)

A estratégia de detecção e recuperação no contexto de deadlocks envolve a verificação periódica do sistema quanto a condições de deadlock usando um algoritmo que identifica condições de espera circular. Uma vez que um deadlock é detectado, o sistema deve se recuperar quebrando o deadlock liberando recursos ou eliminando processos.

As estratégias de recuperação incluem eliminar um ou mais processos em deadlock, antecipar recursos de um ou mais processos ou reverter o progresso de um ou mais processos em deadlock.