Leonardo Vecchi Meirelles - 12031ECP002

Cap 26 - Questão 1/2)

Em um modelo de múltiplos threads, cada thread tem sua própria pilha e contador de programa, mas todos os threads em um processo compartilham o mesmo espaço de endereço virtual, incluindo o mesmo código e os mesmos segmentos de dados.

Isso significa que vários threads podem ler e gravar nos mesmos locais de memória, levando a possíveis condições de race e outros problemas de sincronização.

Cap 26 - Questão 3)

Ao usar threads, existem ganhos como:

- Capacidade de resposta (responsiveness): ao dividir uma tarefa em vários threads, um aplicativo pode ser mais responsivo porque pode continuar processando outras tarefas enquanto espera que um thread conclua seu trabalho
- Compartilhamento de Recursos: threads compartilham o mesmo espaço de memória, o que significa que eles podem facilmente compartilhar dados e se comunicar sem usar mecanismos caros de comunicação entre processos
- Escalabilidade: threads podem ser usados para aproveitar vários processadores ou núcleos, levando a um melhor desempenho
- Modularidade: aplicações podem ser projetadas para ter vários threads, com cada thread responsável por uma tarefa específica, tornando o código mais modular e fácil de manter.
- Eficiência: a criação de threads e a troca de contexto geralmente são mais eficientes do que a criação de processos, permitindo um uso mais eficiente dos recursos do sistema.

Leonardo Vecchi Weixelles - 120 | ECP002

Cap 26 - Questão 4)

Para criar threads usando a biblioteca pthreads, precisamos incluir o arquivo header pthread.h e usar a função pthread_create(). A função recebe quatro parametros: um ponteiro para uma variável pthread_t que armazenará o ID do thread, um ponteiro para uma variável pthread_attr_t que especifica os atributos do thread, um ponteiro para a função que será executada como um thread e um ponteiro para os argumentos que a função recebe.

Para definir o corpo de uma thread, precisamos criar uma função que receba um parâmetro void* e retorne um void*. Esta função conterá o código que será executado como uma thread. Podemos converter o parâmetro para o tipo de dados desejado para acessá-lo dentro da função e devemos retornar NULL para indicar que a thread concluiu sua execução. Uma vez definida a função thread, podemos passá-la como argumento para a função pthread_create() para criar uma nova thread.

Cap 26 - Questão 5)

A ordem de execução de thread é algo não determinístico, mas é possível contornar esse problema. Uma maneira é usar a função pthread_join(), que permite que um thread chamado aguarde a execução de um thread especificado. Outra maneira de controlar a ordem de execução é usando primitivos de sincronização, como mutexes e variáveis de condição. Além disso, pthreads também fornece suporte para políticas de agendamento de threads, que podem ser usados para definir a prioridade de threads e influenciar sua ordem de execução.

Leonardo Vecchi Meirelles - 12011ECP002

Cap 26 - Questão 6)

No contexto de concurrency, uma race condition ocorre quando o comportamento de um programa depende do tempo relativo das operações executadas por dois ou mais threads. Especificamente, uma race condition surge quando vários threads acessam dados compartilhados simultaneamente e pelo menos um dos threads modifica os dados. O comportamento exato do programa depende da ordem em que os threads são executados, o que geralmente é imprevisível e pode levar a bugs. As condições de corrida podem ser difíceis de detectar e depurar, pois podem ocorrer apenas em condições específicas e difíceis de reproduzir.

Cap 26 - Questão 7)

A atomicidade refere-se a uma operação que aparece como se fosse uma única operação indivisível. Uma operação atômica não pode ser interrompida ou intercalada com outras operações, o que garante que a operação seja totalmente ou não executada. Isso é crucial para evitar race conditions e outros problemas de sincronização que podem ocorrer quando vários threads acessam recursos compartilhados.

Cap 27 - Questão 1)

A função pthread_create() recebe quatro argumentos: um ponteiro para uma variável pthread_t, que é usada para identificar exclusivamente o thread recém-criado; um ponteiro para uma estrutura pthread_attr_t, que contém vários atributos que podem ser usados para controlar o comportamento do novo thread; um ponteiro para a função que será executada pelo novo thread; um ponteiro para o argumento que será passado para a função de thread.

Cap 27 - Questão 2)

São dois parâmetros passados para pthread_join(): o parâmetro "thread" especifica o thread a ser unido e "retval" é um ponteiro para um local onde o status de saída do thread unido será armazenado. Se retval não for NULL, então pthread_join() copia o status de saída do thread unida na memória apontada por retval.

A função pthread_join() é bloqueada até que o thread especificado termine. Se a thread já terminou, a função retorna imediatamente. Uma vez que o thread foi unido com sucesso, os recursos por ele utilizados são liberados.

Cap 27 - Questão 3)

Para bloquear um mutex, pode-se usar a função pthread_mutex_lock(), que usa um ponteiro para o mutex como seu argumento. Se o mutex estiver desbloqueado no momento, a função retornará imediatamente e o mutex será bloqueado. Se o mutex já estiver bloqueado por outro thread, a função será bloqueada até que o mutex fique disponível. Depois que o mutex é bloqueado, o thread pode acessar o recurso compartilhado que está protegendo.

Cap 28 - Questão 1)

Um lock é um mecanismo de sincronização que fornece exclusão mútua para um recurso compartilhado. Ele garante que apenas um thread possa manter o bloqueio e acessar o recurso compartilhado por vez, enquanto outros threads que tentam adquirir o lock são bloqueados e colocados em suspensão.

Um lock é representado por um tipo de dados pthread_mutex_t. Para declarar e inicializar um lock, pode-se usar as seguintes funções:

- pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr): inicializa um mutex com os atributos especificados no parametro attr. Se attr for NULL, o mutex será inicializado com os atributos padrão.
- pthread_mutex_destroy(pthread_mutex_t *mutex): destrói um mutex que foi inicializado com pthread_mutex_init().
- pthread_mutex_lock(pthread_mutex_t *mutex): bloqueia o mutex. Se o mutex já estiver bloqueado por outro thread, o thread de chamada será bloqueado até que o mutex seja liberado pelo thread proprietário.
- pthread_mutex_trylock(pthread_mutex_t *mutex): tenta bloquear o mutex, mas se já estiver bloqueado por outro thread, retorna imediatamente com um código de erro.
- pthread_mutex_unlock(pthread_mutex_t *mutex): desbloqueia o mutex, permitindo que outros threads o adquiram.

Cap 28 - Questão 2)

Em geral, permitir que um processo em execução no nível de usuário desabilite as interrupções pode levar a vários problemas. Um dos principais problemas é que pode comprometer a capacidade de resposta do sistema como um todo. Por exemplo, se um processo desabilita as interrupções por um longo período de tempo, outros processos que dependem de interrupções (como operações de I/O) podem sofrer atrasos ou até mesmo não serem concluídos. Além disso, desabilitar as interrupções também pode resultar em problemas de sincronização, como race conditions ou deadlocks, especialmente quando vários processos estão acessando recursos compartilhados.

Permitir que processos desativem interrupções também pode representar um risco de segurança, pois pode permitir que processos maliciosos executem código arbitrário sem interferência. Portanto, a maioria dos sistemas operacionais modernos evita que processos no nível de usuário desativem as interrupções e, em vez disso, contam com outros mecanismos de sincronização, como locks e semáforos, para garantir a exclusão mútua e a coordenação entre processos.

Cap 28 - Questão 3)

Um spin lock é um tipo de bloqueio que opera verificando repetidamente se o lock está disponível, em vez de bloquear quando o lock não está disponível. O lock é mantido em um loop, verificando constantemente até que o lock esteja disponível.

A desvantagem de um spin lock é que ele consome ciclos de CPU enquanto está sendo mantido, mesmo que o thread que mantém o lock não esteja fazendo nenhum progresso. Isso pode levar à redução do desempenho e ao aumento do consumo de recursos. Além disso, os spin locks podem causar inversão de prioridade, em que um thread de alta prioridade é bloqueado por um de prioridade mais baixa que mantém o bloqueio em um loop. Por fim, se o lock for mantido por um longo período de tempo, outros threads que aguardam o lock podem sofrer de starvation.

Cap 28 - Questão 4)

Compare-and-Swap (CAS) é uma instrução atômica comumente utilizada para implementar primitivas de sincronização, como locks, em programas multithread. Ele permite que um thread leia atomicamente um valor da memória, compare-o com um valor esperado e atualize-o condicionalmente para um novo valor se a comparação for bem-sucedida.

Uma desvantagem do CAS é o problema ABA. Isso ocorre quando um thread lê um valor da memória, que subsequentemente muda para outro valor e depois volta para o valor original ("A→B→A"). Se o thread executar uma operação CAS no valor original sem perceber que foi modificado nesse meio tempo, a operação será bem-sucedida, mesmo que outro thread tenha modificado o local da memória nesse meio tempo. Para resolver esse problema, algumas plataformas fornecem uma variante do CAS que usa uma instrução de comparação e troca de largura dupla (DCAS) para comparar e trocar atomicamente dois locais de memória.

Cap 28 - Questão 5)

Load-Linked (LL) e Store-Conditional (SC) são duas instruções fornecidas por alguns processadores para oferecer suporte à sincronização em programação multithread.

LL é utilizado para carregar um local de memória em um registrador e marcar o local como "linked" ao registrador. SC é usado para armazenar um valor de um registro em um local de memória, mas somente se o local ainda estiver "linked" ao mesmo registro. Se a operação de armazenamento falhar, o processador sabe que outro thread modificou o local da memória nesse meio tempo e a operação de armazenamento pode ser repetida.

LL/SC tem algumas limitações, como ser suscetível a problemas de contenção de cache e ordenação de memória em algumas arquiteturas.

Cap 28 - Questão 6)

Fetch-And-Add (FAA) é uma operação atômica que permite incrementar uma variável compartilhada em uma unidade, retornando o valor anterior atomicamente. É uma primitiva de sincronização comumente usada em programação paralela, particularmente em algoritmos sem lock. A operação FAA pode ser implementada usando instruções de hardware, ou usando técnicas de sincronização baseados em software.

A operação FAA consiste em duas etapas: buscar o valor atual da variável compartilhada e, em seguida, adicionar um valor a ela, atualizando seu valor. Durante esta operação, a variável é bloqueada, impedindo que outras threads a acessem. O valor anterior da variável é retornado ao chamador, permitindo que ele execute operações adicionais, se necessário. Por ser atômica, a FAA garante que o valor da variável compartilhada seja atualizado corretamente sem race conditions ou outros problemas de simultaneidade.

Cap 28 - Questão 7)

A primitiva yield() é uma chamada de função que permite que um thread desista voluntariamente de sua vez de executar na CPU. Quando uma thread chama yield(), ela informa ao sistema operacional que está disposta a abrir mão de sua fatia de tempo, permitindo que outra thread seja executada na CPU. O thread que rende pode ou não ser executado novamente imediatamente; depende do algoritmo de agendamento do sistema operacional.

A primitiva yield() é frequentemente usada para evitar busy-waiting ou para melhorar a eficiência do escalonamento de threads. Em vez de um thread verificar repetidamente alguma condição em loop, ele pode ceder a CPU para outro thread que possa ter trabalho a fazer, permitindo que ambos threads progridam. Ao ceder, um thread também pode dar ao escalonador mais opções para decidir qual thread executar em seguida, reduzindo potencialmente o tempo geral de execução.