CS 241: programação do sistema

- atribuições
- Quizzes
- Socorro!
- Cronograma
- Honras
- Funcionários
- Livro didático
- Links

Sincronização

Quando multithreading fica interessante - **Bhuvy**

Sincronização é uma série de mecanismos para controlar quais threads podem executar a operação de cada vez. Na maioria das vezes, os encadeamentos podem progredir sem precisar se comunicar, mas de vez em quando dois ou mais encadeamentos podem querer acessar uma seção crítica. Uma seção crítica é uma seção de código que só pode ser executada por um thread por vez, se o programa funcionar corretamente. Se dois threads (ou processos) executassem o código dentro da seção crítica ao mesmo tempo, é possível que o programa não tenha mais o comportamento correto.

Como dissemos no capítulo anterior, as condições de corrida acontecem quando uma operação toca um pedaço de memória ao mesmo tempo que outro segmento. Se a localização da memória só é acessível por um thread, por exemplo, a variável automática i abaixo, então não há possibilidade de uma condição de corrida e nenhuma Seção Crítica associada a i . No entanto, a variável sum é uma variável global e acessada por dois threads. É possível que dois encadeamentos tentem incrementar a variável ao mesmo tempo.

#include <stdio.h> #include <pthread.h> int sum = 0 ; //shared void * countgold (void * param) { int i ; //local to each thread for (i = 0 ; i < 10000000 ; i ++) { sum += 1 ; } return NULL ; } int main () { pthread_t tid1 , tid2 ; pthread_create (& tid1 , NULL , countgold , NULL); pthread_create (& tid2 , NULL , countgold , NULL); //Wait for both threads to finish: pthread_join (tid1 , NULL); pthread_join (tid2 , NULL); printf ("ARRRRG sum is %d \n " , sum); return 0 ; }

A saída típica do código acima é a ARGGGH sum is <some number less than expected> porque existe uma condição de corrida. O código não pára dois threads da sum de leitura de gravação ao mesmo tempo. Por exemplo, ambos os encadeamentos copiam o valor atual da soma na CPU que executa cada encadeamento (vamos escolher 123). Ambos os threads incrementam um para sua própria cópia. Ambos os threads gravam o valor (124). Se as threads tivessem acessado a soma em momentos diferentes, a contagem teria sido 125. Algumas das possíveis ordenações diferentes estão abaixo.

Padrão admissível

Tópico 1	**Tópico 2**

Carregar Addr, Adicionar 1 (i = 1 localmente)	…
Armazenar (i = 1 globalmente)	…
…	Carregar Addr, Adicionar 1 (i = 2 localmente)
…	Loja (i = 2 globalmente)

Sobreposição parcial

Tópico 1	**Tópico 2**
Carregar Addr, Adicionar 1 (i = 1 localmente)	…
Armazenar (i = 1 globalmente)	Carregar Addr, Adicionar 1 (i = 1 localmente)
…	Armazenar (i = 1 globalmente)

Sobreposição total

Tópico 1	**Tópico 2**
Carregar Addr, Adicionar 1 (i = 1 localmente)	Carregar Addr, Adicionar 1 (i = 1 localmente)
Armazenar (i = 1 globalmente)	Armazenar (i = 1 globalmente)

Gostaríamos que o primeiro padrão do código fosse mutuamente exclusivo. O que nos leva ao nosso primeiro primitivo de sincronização, um mutex.

Mutex

Para garantir que apenas um encadeamento por vez possa acessar uma variável global, use um mutex - abreviação de Exclusão Mútua. Se um thread estiver atualmente dentro de uma seção crítica, gostaríamos que outro thread aguardasse até que o primeiro thread fosse concluído. Um mutex não é um primitivo no sentido mais verdadeiro, embora seja uma das menores partes de estruturas de dados que possui algum sentido de API de threading. Um mutex também não é realmente uma estrutura de dados. É um tipo de dados abstrato, muito parecido com o que você aprendeu em sua classe de estruturas de dados. Há muitas maneiras de implementar um mutex e daremos algumas neste capítulo. Por agora vamos usar a caixa preta que a biblioteca pthread nos dá. Aqui está como nós declaramos um mutex.

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER ; // global variable
pthread_mutex_lock ( & m ); // start of Critical Section // Critical section
pthread_mutex_unlock ( & m ); //end of Critical Section
```

Vida Mutual

Existem algumas maneiras de inicializar um mutex. Você pode usar a macro PTHREAD_MUTEX_INITIALIZER somente para variáveis globais ('estáticas'). m = PTHREAD_MUTEX_INITIALIZER é exatamente equivalente ao propósito mais geral pthread_mutex_init(&m,NULL) mas para nossos propósitos é o mesmo. A versão do init inclui opções para negociar desempenho para verificação adicional de erros e opções avançadas de compartilhamento. A versão init também garante que o mutex seja inicializado corretamente após a chamada, mutexes globais são inicializados no primeiro bloqueio. Você também pode chamar a função init dentro de um programa para um mutex localizado no heap.
pthread_mutex_t * lock = malloc (sizeof (pthread_mutex_t)); pthread_mutex_init (lock , NULL); //later pthread_mutex_destroy (lock); free (lock);
Quando terminarmos o mutex, devemos também chamar pthread_mutex_destroy(&m) também. Note, você só pode destruir um mutex desbloqueado, destruir em um mutex bloqueado é um comportamento indefinido. Coisas que você deve ter em mente sobre o init e o destroy Você não precisa destruir um mutex criado com o inicializador global.

1. Vários threads init / destroy tem comportamento indefinido
2. Destruindo um mutex bloqueado tem comportamento indefinido
3. Mantenha o padrão de um e apenas um thread inicializando um mutex.
4. Copiando os bytes do mutex para um novo local de memória e, em seguida, usando a cópia *não* é suportado. Para fazer referência a um mutex, você precisa ter um ponteiro para esse endereço de memória.

Uso de Mutex

Como eu usaria um mutex? Aqui está um exemplo completo no espírito da peça anterior.
#include <stdio.h> #include <pthread.h> // Create a mutex this ready to be locked! pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER ; int sum = 0 ; void * countgold (void * param) { int i ; //Same thread that locks the mutex must unlock it //Critical section is just 'sum += 1' //However locking and unlocking a million times //has significant overhead pthread_mutex_lock (& m); // Other threads that call lock will have to wait until we call unlock for (i = 0 ; i < 10000000 ; i ++) { sum += 1 ; } pthread_mutex_unlock (& m); return NULL ; } int main () { pthread_t tid1 , tid2 ; pthread_create (& tid1 , NULL , countgold , NULL); pthread_create (& tid2 , NULL , countgold , NULL); pthread_join (tid1 , NULL); pthread_join (tid2 , NULL); printf ("ARRRRG sum is %d \n " , sum); return 0 ; }
No código acima, o thread recebe o bloqueio para a casa de contagem antes de entrar. A seção crítica é apenas a sum+=1 portanto, a versão a seguir também está correta.
for (i = 0 ; i < 10000000 ; i ++) { pthread_mutex_lock (& m); sum += 1 ; pthread_mutex_unlock (& m); } return NULL ; }
Esse processo é executado mais devagar porque bloqueamos e desbloqueamos o mutex um milhão de vezes, o que é caro - pelo menos em comparação com o incremento de uma variável. Neste exemplo simples, nós realmente não precisamos de threads - poderíamos ter adicionado duas vezes! Um exemplo mais rápido de multi-thread seria adicionar um milhão usando uma variável automática (local) e só depois adicioná-lo a um total compartilhado após o término do ciclo de cálculo:
int local = 0 ; for (i = 0 ; i < 10000000 ; i ++) { local += 1 ; } pthread_mutex_lock (& m); sum += local ; pthread_mutex_unlock (& m); return NULL ; }

Obviamente, se você conhece a soma gaussiana, pode evitar completamente as condições da corrida, mas isso é apenas para ilustrar.
Começando com as pegadinhas. Em primeiro lugar, os Mutexes C não bloqueiam variáveis. Um mutex não é tão inteligente. Funciona com código, não dados. Se eu bloquear um mutex, os outros threads continuarão. É somente quando um thread tenta bloquear um mutex que já está bloqueado, o thread terá que esperar. Assim que o thread original desbloqueia o mutex, o segundo segmento (em espera) adquirirá o bloqueio e poderá continuar. O código a seguir cria um mutex que efetivamente não faz nada.
int a ; pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER , m2 = = PTHREAD_MUTEX_INITIALIZER ; // later // Thread 1 pthread_mutex_lock (& m1); a ++ ; pthread_mutex_unlock (& m1); // Thread 2 pthread_mutex_lock (& m2); a ++ ; pthread_mutex_unlock (& m2);
Aqui estão algumas outras dicas em nenhuma ordem particular

1. Não cruze os fluxos! Se você estiver usando threads, não bifurque no meio do seu programa. Isso significa que qualquer hora depois de seus mutexes terem sido inicializados.
2. O thread que bloqueia um mutex é o único segmento que pode desbloqueá-lo.
3. Cada programa pode ter vários bloqueios mutex. Você pode projetar seu programa para ter um bloqueio por estrutura de dados, um bloqueio por heap ou um bloqueio por conjunto de estruturas de dados. Se você tiver apenas um bloqueio, eles podem ser uma disputa significativa para o bloqueio entre dois segmentos desnecessários. Por exemplo, se dois encadeamentos estiverem atualizando dois contadores diferentes, pode não ser necessário usar o mesmo bloqueio.
4. Bloqueios são apenas ferramentas. Eles não localizam seções críticas para você!
5. Sempre haverá uma pequena sobrecarga de chamadas pthread_mutex_lock e pthread_mutex_unlock . No entanto, este é o preço que você paga por programas funcionando corretamente!
6. Não desbloquear um mutex devido a um retorno antecipado durante uma condição de erro
7. Vazamento de recursos (não chamando pthread_mutex_destroy)
8. Usando um mutex não inicializado ou usando um mutex que já foi destruído
9. Bloquear um mutex duas vezes em um thread sem desbloquear primeiro
10. Impasse

Implementação de Mutex

Então nós temos essa estrutura de dados legal. Como podemos implementá-lo? Primeiro, uma implementação ingênua e incorreta é mostrada abaixo. A função de unlock simplesmente desbloqueia o mutex e retorna. A função de bloqueio primeiro verifica se a trava já está bloqueada. Se estiver bloqueado no momento, ele continuará sendo verificado até que outro thread tenha desbloqueado o mutex.
// Version 1 (Incorrect!) void lock (mutex_t * m) { while (m -> locked) { /*Locked? Never-mind - just loop and check again!*/ } m -> locked = 1 ; } void unlock (mutex_t * m) { m -> locked = 0 ; }
A versão 1 usa 'espera ocupada' desnecessariamente desperdiçando recursos da CPU. No entanto, há um problema mais sério. Nós temos uma condição de corrida! Se dois

encadeamentos forem chamados simultaneamente, é possível que ambos os encadeamentos leiam m_locked como zero. Assim, os dois segmentos acreditarão que terão acesso exclusivo ao bloqueio e os dois segmentos continuarão.
Podemos tentar reduzir um pouco a sobrecarga da CPU chamando pthread_yield() dentro do loop - o pthread_yield sugere ao sistema operacional que o encadeamento não usa a CPU por um curto período de tempo, então a CPU pode ser atribuída a encadeamentos que estão esperando para corre. Mas não corrige a condição de corrida. Precisamos de uma implementação melhor.

Extra: Implementando um mutex com hardware

Podemos usar o C11 Atomics para fazer isso perfeitamente! Uma solução completa é detalhada aqui. Este é um mutex spinlock, https://locklessinc.com/articles/mutex_cv_futex/ implementações podem ser encontradas on-line.
Primeiro a estrutura de dados e o código de inicialização.
typedef struct mutex_ { // We need some variable to see if the lock is locked atomic_int_least8_t lock ; // A mutex needs to keep track of its owner so // Another thread can't unlock it pthread_t owner ; } mutex ; #define UNLOCKED 0 #define LOCKED 1 #define UNASSIGNED_OWNER 0 int mutex_init (mutex * mtx){ // Some simple error checking if (! mtx){ return 0 ; } // Not thread safe the user has to take care of this atomic_init (& mtx -> lock , UNLOCKED); mtx -> owner = UNASSIGNED_OWNER ; return 1 ; }
Este é o código de inicialização, nada extravagante aqui. Nós definimos o estado do mutex como desbloqueado e definimos o proprietário como bloqueado.
int mutex_lock (mutex * mtx){ int_least8_t zero = UNLOCKED ; while (! atomic_compare_exchange_weak_explicit (& mtx -> lock , & zero , LOCKED , memory_order_seq_cst , memory_order_seq_cst)){ zero = UNLOCKED ; sched_yield (); // Use system calls for scheduling speed } // We have the lock now mtx -> owner = pthread_self (); return 1 ; }
O que faz este código? Bem, para começar, inicializa uma variável que manteremos como o estado desbloqueado. https://en.wikipedia.org/wiki/Compare-and-swap é uma instrução suportada pela maioria das arquiteturas modernas (no x86 é lock cmpxchg). O pseudocódigo desta operação é semelhante a este
int atomic_compare_exchange_pseudo (int * addr1 , int * addr2 , int val){ if (* addr1 == * addr2){ * addr1 = val ; return 1 ; } else { * addr2 = * addr1 ; return 0 ; } }
Exceto que tudo é feito *atomicamente,* ou seja, em uma operação ininterrupta. O que significa a parte *fraca* ? Bem instruções atômicas são propensas a **falhas espúrias,** o que significa que há duas versões para essas funções atômicas, uma *forte* e uma parte *fraca* , forte garante o sucesso ou fracasso, enquanto fraco pode falhar, mesmo quando a operação for bem-sucedida. Estas são as mesmas falhas espúrias que você verá nas variáveis de condição abaixo. Estamos usando fraco porque fraco é mais rápido, e estamos em um loop! Isso significa que estamos bem se falhar um pouco mais, porque vamos continuar girando de qualquer maneira.
Dentro do loop while, não conseguimos pegar o bloqueio! Nós zeramos o zero para destravar e dormimos por um tempo. Quando acordamos, tentamos pegar a fechadura novamente. Uma vez que trocamos com sucesso, estamos na seção crítica! Nós configuramos o proprietário do mutex para o segmento atual para o método de desbloqueio e retornamos com sucesso.

Como isso garante a exclusão mútua, quando trabalhamos com átomos, não temos certeza absoluta! Mas neste exemplo simples, podemos porque o thread que é capaz de esperar com sucesso que o bloqueio seja UNLOCKED (0) e trocá-lo para um estado LOCKED (1) é considerado o vencedor. Como implementamos o desbloqueio?
int mutex_unlock (mutex * mtx){ if (unlikely (pthread_self () != mtx -> owner)){ return 0 ; //You can't unlock a mutex if you aren't the owner } int_least8_t one = 1 ; //Critical section ends after this atomic mtx -> owner = UNASSIGNED_OWNER ; if (! atomic_compare_exchange_strong_explicit (& mtx -> lock , & one , UNLOCKED , memory_order_seq_cst , memory_order_seq_cst)){ //The mutex was never locked in the first place return 0 ; } return 1 ; }
Para satisfazer a API, você não pode desbloquear o mutex, a menos que você seja o dono dele. Em seguida, desatribuímos o proprietário do mutex, porque a seção crítica termina após o atômico. Queremos uma troca forte porque não queremos bloquear (o pthread_mutex_unlock não bloqueia). Esperamos que o mutex seja bloqueado e nós o trocamos para desbloquear. Se a troca foi bem sucedida, nós desbloqueamos o mutex. Se a troca não foi, isso significa que o mutex foi UNLOCKED e nós tentamos mudá-lo de UNLOCKED para UNLOCKED, preservando o bloqueio não de desbloqueio.
O que é esse negócio de pedidos de memória? Nós estávamos falando sobre cercas de memória mais cedo, aqui está! Não entraremos em detalhes porque está fora do escopo deste curso, mas não no escopo do https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync . Basicamente, precisamos de consistência para garantir que nenhuma carga ou armazenamento seja encomendada antes ou depois. Você precisa criar cadeias de dependência para pedidos mais eficientes.

Semáforo

Um semáforo é outro primitivo de sincronização. É inicializado com algum valor. Os threads podem sem_wait ou sem_post que diminui ou aumenta o valor. Se o valor chegar a zero e uma espera for chamada, o encadeamento será bloqueado até que uma postagem seja chamada.
Usar um semáforo é tão fácil quanto criar um mutex. Primeiro, decida se o valor inicial deve ser zero ou algum outro valor, por exemplo, o número de espaços restantes em uma matriz. Ao contrário do pexhar mutex, não há atalhos para criar um semáforo - use sem_init .
#include <semaphore.h> sem_t s ; int main () { sem_init (& s , 0 , 10); // returns -1 (=FAILED) on OS X sem_wait (& s); // Could do this 10 times without blocking sem_post (& s); // Announce that we've finished (and one more resource item is available; increment count) sem_destroy (& s); // release resources of the semaphore }
Ao usar um semáforo, esperar e postar podem ser chamados de diferentes threads! Ao contrário de um mutex, o incremento e o decremento podem ser de diferentes segmentos. Isso se torna especialmente útil se você quiser usar um semáforo para implementar um mutex. Um mutex é um semáforo que sempre waits antes de ser posts . Alguns livros se referem a um mutex como um semáforo binário. Você precisa ter cuidado para nunca adicionar mais de um a um semáforo ou, caso contrário, a abstração do mutex será interrompida. Geralmente é por isso que um mutex é usado para implementar um semáforo e vice-versa.

- Inicialize o semáforo com uma contagem de um.

- Substitua pthread_mutex_lock por sem_wait
- Substitua pthread_mutex_unlock por sem_post

sem_t s ; sem_init (& s , 0 , 1); sem_wait (& s); // Critical Section sem_post (& s);
Mas esteja avisado, não é o mesmo! Um mutex pode lidar com o que chamamos de inversão de bloqueio bem. O que significa o seguinte código quebra com um mutex tradicional, mas produz uma condição de corrida com threads.
// Thread 1 sem_wait (& s); // Critical Section sem_post (& s); // Thread 2 // Some threads just want to see the world burn sem_post (& s); // Thread 3 sem_wait (& s); // Not thread safe! sem_post (& s);
Se substituirmos por bloqueio mutex, isso não funcionará agora.
// Thread 1 mutex_lock (& s); // Critical Section mutex_unlock (& s); // Thread 2 // Foiled! mutex_unlock (& s); // Thread 3 mutex_lock (& s); // Now it's thread safe mutex_unlock (& s);

Segurança de Sinal

Além disso, a palavra-chave sem_post é uma das poucas funções que podem ser usadas corretamente dentro de um manipulador de sinal pthread_mutex_unlock . Isso significa que podemos liberar um thread em espera que agora pode fazer todas as chamadas que não pudemos chamar dentro do próprio manipulador de sinal, por exemplo, printf . Aqui está algum código que utiliza isso;
#include <stdio.h> #include <pthread.h> #include <signal.h> #include <semaphore.h> #include <unistd.h> sem_t s ; void handler (int signal) { sem_post (& s); /* Release the Kraken! */ } void * singsong (void * param) { sem_wait (& s); printf ("I had to wait until your signal released me! \n "); } int main () { int ok = sem_init (& s , 0 , 0 /* Initial value of zero*/); if (ok == - 1) { perror ("Could not create unnamed semaphore"); return 1 ; } signal (SIGINT , handler); // Too simple! See Signals chapter pthread_t tid ; pthread_create (& tid , NULL , singsong , NULL); pthread_exit (NULL); /* Process will exit when there are no more threads */ }
Outros usos para semáforos estão mantendo o controle de espaços vazios em matrizes. Discutiremos isso na seção de estruturas de dados seguras para thread.

Variáveis de condição

As variáveis de condição permitem que um conjunto de encadeamentos durem até que o woken seja ativado. Você pode acordar um thread ou todos os threads que estão dormindo. Se você ativar apenas um thread, o sistema operacional decidirá qual thread será ativado. Você não ativa as threads diretamente, em vez disso, "sinaliza" a variável de condição, que então desperta um (ou todos) threads que estão dormindo dentro da variável de condição. As variáveis de condição também são usadas com um mutex e com um loop, portanto, quando acordadas, elas precisam verificar uma condição em uma seção crítica. Se você só precisa ser acordado não em uma seção crítica, existem outras maneiras de fazer isso no POSIX. Threads dormindo dentro de uma variável de condição são ativados chamando pthread_cond_broadcast (wake up all) ou pthread_cond_signal (wake up one). Nota apesar do nome da função, isso não tem nada a ver com o signal POSIX!

Ocasionalmente, um fio de espera pode parecer acordar sem motivo. Isso é chamado de *vigília espúria* . Se você ler a implementação de hardware de uma seção mutex, isso é semelhante. Isso não é um problema porque você sempre usa wait dentro de um loop que testa uma condição que deve ser verdadeira para continuar.
Por que surtos espúrios acontecem? Para desempenho. Em sistemas com várias CPUs, é possível que uma condição de corrida possa fazer com que uma solicitação de despertar (sinal) não seja notada. O kernel pode não detectar essa chamada de ativação perdida, mas pode detectar quando ela pode ocorrer. Para evitar o potencial sinal perdido, o encadeamento é ativado para que o código do programa possa testar a condição novamente.

Condição de espera Wait Example

A chamada pthread_cond_wait executa três ações:

1. Desbloqueie o mutex; deve estar trancado.
2. Dorme até que pthread_cond_signal seja chamado na mesma variável de condição)
3. Antes de retornar, bloqueia o mutex

As variáveis de condição são *sempre* usadas com um bloqueio mutex. Antes de chamar *wait* , o bloqueio mutex deve ser bloqueado e a *espera* deve ser envolvida por um loop. pthread_cond_t cv ; pthread_mutex_t m ; int count ; // Initialize pthread_cond_init (& cv , NULL); pthread_mutex_init (& m , NULL); count = 0 ; // Thread 1 pthread_mutex_lock (& m); while (count < 10) { pthread_cond_wait (& cv , & m); /* Remember that cond_wait unlocks the mutex before blocking (waiting)! */ /* After unlocking, other threads can claim the mutex. */ /* When this thread is later woken it will */ /* re-lock the mutex before returning */ } pthread_mutex_unlock (& m); //later clean up with pthread_cond_destroy(&cv); and mutex_destroy // Thread 2: while (1) { pthread_mutex_lock (& m); count ++ ; pthread_cond_signal (& cv); /* Even though the other thread is woken up it cannot not return */ /* from pthread_cond_wait until we have unlocked the mutex. This is */ /* a good thing! In fact, it is usually the best practice to call */ /* cond_signal or cond_broadcast before unlocking the mutex */ pthread_mutex_unlock (& m); }
Este é um exemplo bastante ingênuo, mas mostra que podemos dizer que os tópicos sejam ativados de maneira padronizada. Na próxima seção, usaremos estes para implementar estruturas de dados de bloqueio eficientes.

Estruturas de dados seguras de thread

Naturalmente, queremos que nossas estruturas de dados também sejam seguras para threads! Podemos usar mutexes e primitivos de sincronização para que isso aconteça. Primeiro algumas definições. Atomicidade é quando uma operação é thread-safe. Nós temos instruções atômicas em hardware, fornecendo o prefixo de bloqueio
lock ...
Mas Atomicity também se aplica a ordens mais altas de operações. Dizemos que uma operação de estrutura de dados é atômica se acontecer de uma vez e com sucesso ou não. Como tal, podemos usar primitivas de sincronização para tornar nossas estruturas de dados seguras. Na maioria das vezes, usaremos mutexes porque eles carregam mais significado

semântico que um semáforo binário. Note que esta é apenas uma introdução - escrever estruturas de dados seguras para segmentos de alto desempenho requer seu próprio livro! Tomemos por exemplo a seguinte pilha não thread-safe.

```
// A simple fixed-sized stack (version 1) #define STACK_SIZE 20 int count ; double values [ STACK_SIZE ]; void push ( double v ) { values [ count ++ ] = v ; } double pop () { return values [ -- count ]; } int is_empty () { return count == 0 ; }
```

A versão 1 da pilha não é thread-safe porque, se duas linhas chamarem push ou pop ao mesmo tempo, os resultados ou a pilha poderão ficar inconsistentes. Por exemplo, imagine se dois encadeamentos chamam pop ao mesmo tempo, então ambos os encadeamentos podem ler o mesmo valor, ambos podem ler o valor da contagem original.

Para transformar isso em uma estrutura de dados segura para encadeamentos, precisamos identificar as *seções críticas* do nosso código, o que significa que precisamos perguntar qual (is) seção (s) do código deve ter apenas um encadeamento por vez. No exemplo acima, pop funções push , pop e is_empty acessam a mesma memória e todas as seções críticas da pilha. Enquanto push (e pop) está sendo executado, a estrutura de dados é um estado inconsistente, por exemplo, a contagem pode não ter sido gravada, portanto, ainda pode conter o valor original. Envolvendo esses métodos com um mutex, podemos garantir que apenas um thread de cada vez possa atualizar (ou ler) a pilha. Uma 'solução' candidata é mostrada abaixo. Está correto? Se não, como isso falhará?

```
// An attempt at a thread-safe stack (version 2) #define STACK_SIZE 20 int count ; double values [ STACK_SIZE ]; pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER ; pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER ; void push ( double v ) { pthread_mutex_lock ( & m1 ); values [ count ++ ] = v ; pthread_mutex_unlock ( & m1 ); } double pop () { pthread_mutex_lock ( & m2 ); double v = values [ -- count ]; pthread_mutex_unlock ( & m2 ); return v ; } int is_empty () { pthread_mutex_lock ( & m1 ); return count == 0 ; pthread_mutex_unlock ( & m1 ); }
```

A versão 2 contém pelo menos um erro. Tome um momento para ver se você pode o (s) erro (s) e elaborar as conseqüências.

Se três encadeamentos chamados push() ao mesmo tempo, o bloqueio m1 garante que apenas um encadeamento no momento manipule a pilha em push ou is_empty - Dois encadeamentos precisarão aguardar até que o primeiro encadeamento seja concluído Um argumento semelhante se aplica a chamadas simultâneas para pop . No entanto, a versão 2 não impede que push e pop sejam executados ao mesmo tempo, porque push e pop usam dois bloqueios mutex diferentes. A correção é simples neste caso - use o mesmo bloqueio mutex para as funções push e pop.

O código tem um segundo erro. is_empty retorna após a comparação e não desbloqueará o mutex. No entanto, o erro não seria detectado imediatamente. Por exemplo, suponha que um encadeamento chame is_empty e um segundo encadeamento depois chame push . Este fio iria parar misteriosamente. Usando o depurador, você pode descobrir que o encadeamento está preso no método lock () dentro do método push porque o bloqueio nunca foi desbloqueado pela chamada is_empty anterior. Assim, um descuido em um segmento levou a problemas muito mais tarde, em um outro segmento arbitrário. Vamos tentar corrigir esses problemas

```
// An attempt at a thread-safe stack (version 3) int count ; double values [ count ]; pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER ; void push ( double v ) { pthread_mutex_lock ( & m ); values [ count ++ ] = v ; pthread_mutex_unlock ( & m ); } double
```

pop () { pthread_mutex_lock (& m); double v = values [-- count]; pthread_mutex_unlock (& m); return v ; } int is_empty () { pthread_mutex_lock (& m); int result = count == 0 ; pthread_mutex_unlock (& m); return result ; }
Versão 3 é thread-safe. Garantimos a exclusão mútua de todas as seções críticas. Há algumas coisas a serem observadas.

- is_empty é thread-safe, mas seu resultado pode já estar desatualizado. A pilha pode não estar mais vazia no momento em que o encadeamento obtém o resultado! Geralmente, isso ocorre porque, nas estruturas de dados seguras de thread, as funções que retornam tamanhos são removidas ou preteridas.
- Não há proteção contra estouro negativo (estouro em uma pilha vazia) ou estouro (empurrando para uma pilha já cheia)

O último ponto pode ser corrigido usando semáforos de contagem. A implementação assume uma única pilha. Uma versão de propósito mais geral pode incluir o mutex como parte da estrutura da memória e usar pthread_mutex_init para inicializar o mutex. Por exemplo,
// Support for multiple stacks (each one has a mutex) typedef struct stack { int count ; pthread_mutex_t m ; double * values ; } stack_t ; stack_t * stack_create (int capacity) { stack_t * result = malloc (sizeof (stack_t)); result -> count = 0 ; result -> values = malloc (sizeof (double) * capacity); pthread_mutex_init (& result -> m , NULL); return result ; } void stack_destroy (stack_t * s) { free (s -> values); pthread_mutex_destroy (& s -> m); free (s); } // Warning no underflow or overflow checks! void push (stack_t * s , double v) { pthread_mutex_lock (& s -> m); s -> values [(s -> count) ++] = v ; pthread_mutex_unlock (& s -> m); } double pop (stack_t * s) { pthread_mutex_lock (& s -> m); double v = s -> values [-- (s -> count)]; pthread_mutex_unlock (& s -> m); return v ; } int is_empty (stack_t * s) { pthread_mutex_lock (& s -> m); int result = s -> count == 0 ; pthread_mutex_unlock (& s -> m); return result ; } int main () { stack_t * s1 = stack_create (10 /* Max capacity*/); stack_t * s2 = stack_create (10); push (s1 , 3 . 141); push (s2 , pop (s1)); stack_destroy (s2); stack_destroy (s1); }
Antes de consertarmos os problemas com semáforos. Como podemos corrigir os problemas com variáveis de condição? Experimente antes de ver o código na seção anterior.
Basicamente, precisamos esperar em push e pop se nossa pilha estiver cheia ou vazia, respectivamente. Tentativa de solução:
// Assume cv is a condition variable // correctly initialized void push (stack_t * s , double v) { pthread_mutex_lock (& s -> m); if (s -> count == 0) pthread_cond_wait (& s -> cv , & s -> m); s -> values [(s -> count) ++] = v ; pthread_mutex_unlock (& s -> m); } double pop (stack_t * s) { pthread_mutex_lock (& s -> m); if (s -> count == 0) pthread_cond_wait (& s -> cv , & s -> m); double v = s -> values [-- (s -> count)]; pthread_mutex_unlock (& s -> m); return v ; }
A solução a seguir funciona? Tome um segundo antes de olhar para a resposta para identificar os erros.
Então você pegou todos eles?

1. O primeiro é simples. Em push, nossa verificação deve ser contra a capacidade total e não zero.
2. Nós só temos verificações de instrução. wait () poderia despertar falsamente

3. Nós nunca sinalizamos nenhum dos tópicos! Segmentos podem ficar presos indefinidamente.

Vamos consertar esses erros Esta solução funciona?
void push (stack_t * s , double v) { pthread_mutex_lock (& s -> m); while (s -> count == capacity) pthread_cond_wait (& s -> cv , & s -> m); s -> values [(s -> count) ++] = v ; pthread_cond_signal (& s -> cv); pthread_mutex_unlock (& s -> m); } double pop (stack_t * s) { pthread_mutex_lock (& s -> m); while (s -> count == 0) pthread_cond_wait (& s -> cv , & s -> m); double v = s -> values [-- (s -> count)]; pthread_cond_broadcast (& s -> cv); pthread_mutex_unlock (& s -> m); return v ; }
Esta solução também não funciona! O problema é com o sinal. Você pode ver por quê? O que você faria para consertar isso?
Agora, como é que vamos usar os semáforos para evitar over e underflow? Vamos discutir isso na próxima seção.

Usando semáforos

Vamos usar um semáforo de contagem para controlar quantos espaços permanecem e outro semáforo para rastrear o número de itens na pilha. Nós chamaremos esses dois semáforos sremain e sitems . Lembre-se que sem_wait irá esperar se a contagem do semáforo foi diminuída para zero (por outro thread chamando sem_post).
// Sketch #1 sem_t sitems ; sem_t sremain ; void stack_init (){ sem_init (& sitems , 0 , 0); sem_init (& sremain , 0 , 10); } double pop () { // Wait until there's at least one item sem_wait (& sitems); ... void push (double v) { // Wait until there's at least one space sem_wait (& sremain); ... }
O esboço # 2 implementou o post cedo demais. Outro thread esperando no push pode erroneamente tentar escrever em uma pilha completa e, da mesma forma, um thread esperando no pop () pode continuar muito cedo.
// Sketch #2 (Error!) double pop () { // Wait until there's at least one item sem_wait (& sitems); sem_post (& sremain); // error! wakes up pushing() thread too early return values [-- count]; } void push (double v) { // Wait until there's at least one space sem_wait (& sremain); sem_post (& sitems); // error! wakes up a popping() thread too early values [count ++] = v ; }
O Sketch 3 implementa a lógica correta do semáforo, mas você consegue identificar o erro?
// Sketch #3 (Error!) double pop () { // Wait until there's at least one item sem_wait (& sitems); double v = values [-- count]; sem_post (& sremain); return v ; } void push (double v) { // Wait until there's at least one space sem_wait (& sremain); values [count ++] = v ; sem_post (& sitems); }
O Sketch 3 aplica corretamente as condições de buffer full e buffer empty usando semáforos. No entanto, não há *exclusão mútua* . Dois segmentos podem estar na *seção crítica* ao mesmo tempo, o que corromperia a estrutura de dados ou levaria à perda de dados. A correção é envolver um mutex em torno da seção crítica:
// Simple single stack - see above example on how to convert this into a multiple stacks. // Also a robust POSIX implementation would check for EINTR and error codes of sem_wait. // PTHREAD_MUTEX_INITIALIZER for statics (use pthread_mutex_init() for stack/heap memory) #define SPACES 10 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER ; int count = 0 ; double values [SPACES]; sem_t sitems , sremain ; void init () { sem_init (&

sitems , 0 , 0); sem_init (& sremains , 0 , SPACES); // 10 spaces } double pop () { // Wait until there's at least one item sem_wait (& sitems); pthread_mutex_lock (& m); // CRITICAL SECTION double v = values [-- count]; pthread_mutex_unlock (& m); sem_post (& sremain); // Hey world, there's at least one space return v ; } void push (double v) { // Wait until there's at least one space sem_wait (& sremain); pthread_mutex_lock (& m); // CRITICAL SECTION values [count ++] = v ; pthread_mutex_unlock (& m); sem_post (& sitems); // Hey world, there's at least one item } // Note a robust solution will need to check sem_wait's result for EINTR (more about this later)

Agora vamos começar a fazer algumas coisas malucas. O que acontece quando começamos a inverter o bloqueio e esperar ordens?

double pop () { pthread_mutex_lock (& m); sem_wait (& sitems); double v = values [-- count]; pthread_mutex_unlock (& m); sem_post (& sremain); return v ; } void push (double v) { sem_wait (& sremain); pthread_mutex_lock (& m); values [count ++] = v ; pthread_mutex_unlock (& m); sem_post (& sitems); }

Em vez de apenas dar a resposta, deixaremos você pensar sobre isso. Esta é uma maneira permitida de bloquear e desbloquear? Existe uma série de padrões que podem causar uma condição de corrida? Como sobre impasse? Se houver, forneça. Se não houver, forneça uma breve prova em inglês de por que isso não acontecerá.

Soluções de Software para a Seção Crítica

Como já foi discutido, existem partes críticas do nosso código que só podem ser executadas por um thread de cada vez. Descrevemos este requisito como "exclusão mútua". Apenas um thread (ou processo) pode ter acesso ao recurso compartilhado. Em programas multi-threaded, podemos envolver uma seção crítica com bloqueio mutex e desbloquear chamadas:

pthread_mutex_lock () // one thread allowed at a time! (others will have to wait here) // ... Do Critical Section stuff here! pthread_mutex_unlock () // let other waiting threads continue

Como podemos implementar essas chamadas de bloqueio e desbloqueio? Podemos criar um algoritmo de software puro que garanta a exclusão mútua?

pthread_mutex_lock (p_mutex_t * m) { while (m -> lock) ; m -> lock = 1 ; } pthread_mutex_unlock (p_mutex_t * m) { m -> lock = 0 ; }

À primeira vista, o código parece funcionar. Se um thread tentar bloquear o mutex, um thread posterior deve aguardar até que o bloqueio seja limpo. No entanto, esta implementação *não satisfaz a Exclusão Mútua* . Ignoraremos a parte sobre outros threads sendo capazes de desbloquear o bloqueio de outro segmento. Vamos dar uma olhada de perto nessa 'implementação' do ponto de vista de dois threads rodando ao mesmo tempo. Para simplificar a discussão, consideramos apenas dois tópicos. Observe que esses argumentos funcionam para encadeamentos e processos, e a literatura clássica de CS discute esses problemas em termos de dois processos que precisam de acesso exclusivo a uma seção crítica ou a um recurso compartilhado. Levantar uma bandeira representa a intenção de um processo / thread de entrar na seção crítica.

Lembre-se de que o pseudo-código descrito abaixo é parte de um programa maior. O encadeamento ou processo normalmente precisará entrar na seção crítica muitas vezes durante a vida útil do processo. Portanto, imagine cada exemplo como envolvido em um

loop onde, por um período de tempo aleatório, o thread ou processo está trabalhando em outra coisa.
Há algo de errado com a solução candidata descrita abaixo?
// Candidate #1 wait until your flag is lowered raise my flag // Do Critical Section stuff lower my flag
Resposta: A solução nº 1 do candidato também sofre uma condição de disputa, pois os dois processos / encadeamentos podem ler o valor do sinalizador um do outro, conforme diminuem, e continuam.
Isto sugere que devemos levantar o sinalizador *antes de* verificar o sinalizador do outro segmento, que é a solução candidata # 2 abaixo.
// Candidate #2 raise my flag wait until your flag is lowered // Do Critical Section stuff lower my flag
O candidato nº 2 satisfaz a exclusão mútua - é impossível que dois segmentos estejam dentro da seção crítica ao mesmo tempo. No entanto, este código sofre de deadlock! Suponha que dois segmentos desejem entrar na seção crítica ao mesmo tempo.

Tempo	**Tópico 1**	**Tópico 2**
1	Levantar bandeira	
2		Levantar bandeira
3	Esperar	Esperar

Ambos os processos / threads estão agora aguardando que o outro diminua seus sinalizadores. Nenhum deles entrará na seção crítica, pois ambos estão presos para sempre! Isso sugere que devemos usar uma variável baseada em turnos para tentar resolver quem deve prosseguir.

Soluções baseadas em turnos

A seguinte solução candidata nº 3 usa uma variável baseada em turnos para permitir educadamente que um thread e o outro continuem
// Candidate #3 wait until my turn is myid // Do Critical Section stuff turn = yourid
O candidato nº 3 satisfaz a exclusão mútua - cada thread ou processo obtém acesso exclusivo à Seção Crítica. No entanto, os dois segmentos / processos devem ter uma abordagem estrita baseada em turnos para usar a seção crítica. Eles são forçados a um padrão alternativo de acesso de seção crítica. Por exemplo, se o thread 1 desejar ler uma tabela de hash a cada milissegundo, mas outro thread gravar em uma tabela de hash a cada segundo, o thread de leitura teria que esperar outros 999ms antes de poder ler a tabela de hash novamente. Essa 'solução' não é eficaz porque nossos threads devem poder progredir e entrar na seção crítica se nenhum outro thread estiver atualmente na seção crítica.

Propriedades desejadas para soluções

Existem três propriedades desejáveis principais que desejamos em uma solução o problema da seção crítica

1. Exclusão mútua. O segmento / processo obtém acesso exclusivo. Outros devem esperar até que saia da seção crítica.
2. Espera Limitada. Um encadeamento / processo não pode ser substituído por outro encadeamento por um período infinito de tempo.
3. Progresso. Se nenhum thread / processo estiver dentro da seção crítica, o thread / processo deve poder prosseguir sem ter que esperar.

Com essas ideias em mente, vamos examinar outra solução candidata que use um sinalizador baseado em turnos apenas se dois segmentos precisarem de acesso ao mesmo tempo.

Soluções de virada e sinalização

O seguinte é uma solução correta para o CSP?
\\ Candidate #4 raise my flag if your flag is raised, wait until my turn // Do Critical Section stuff turn = yourid lower my flag
Um instrutor e um outro membro da faculdade CS inicialmente pensava assim TODO: **carece de fontes** ! No entanto, analisar essas soluções é complicado. Mesmo os artigos revisados por especialistas sobre este assunto específico contêm soluções incorretas! À primeira vista, parece satisfazer Exclusão Mútua, Espera Contínua e Progresso. A bandeira baseada em turnos é usada apenas no caso de empate, portanto Progresso e Espera Limitada são permitidos e a exclusão mútua parece ser satisfeita. Talvez você possa encontrar um contra-exemplo?
O candidato nº 4 falha porque um segmento não espera até que o outro segmento abaixe seu sinalizador. Após algum pensamento ou inspiração, o seguinte cenário pode ser criado para demonstrar como a Exclusão Mútua não é satisfeita.
Imagine o primeiro segmento executa este código duas vezes. O indicador de turno agora aponta para o segundo segmento. Enquanto o primeiro segmento ainda está dentro da seção Crítica, o segundo segmento chega. O segundo segmento pode continuar imediatamente na seção crítica!

Tempo	Virar	Tópico 1	Tópico 2
1	2	Levante minha bandeira	
2	2	Se a sua bandeira for levantada, espere até a minha vez	Levante minha bandeira
3	2	// Fazer Material da Seção Crítica	Se a sua bandeira for levantada, espere até a minha vez (TRUE!)
4	2	// Fazer Material da Seção Crítica	Faça coisas críticas da seção - OOPS

Soluções de trabalho

A primeira solução para o problema foi a Solução Dekker. O Algoritmo de Dekker (1962) foi a primeira solução comprovadamente correta. No entanto, foi em um artigo não publicado, por isso não foi descoberto até mais tarde (Dekker e Dijkstra # ref-dekker_dijkstra_1965) - esta é uma versão transcrita Inglês lançado em 1965. Uma versão do algoritmo é abaixo. raise my flag while (your flag is raised) : if it's your turn to win : lower my flag wait while your turn raise my flag // Do Critical Section stuff set your turn to win lower my flag
Observe como o sinalizador do processo é sempre levantado durante a seção crítica, não importa se o loop é iterado zero, uma ou mais vezes. Além disso, a bandeira pode ser interpretada como uma intenção imediata de entrar na seção crítica. Somente se o outro processo também tiver levantado o sinalizador, um processo será adiado, abaixe o sinalizador de intenção e espere.
Vamos verificar as condições.

1. Exclusão mútua. Vamos tentar esboçar uma prova simples. O loop invariante é que, no início da verificação da condição, sua bandeira precisa ser levantada - isso é por exaustão. Como a única maneira que um thread pode deixar o loop é ter a condição false, o sinalizador deve ser elevado para a totalidade da seção crítica. Como o loop impede a saída de um thread enquanto o flag do outro thread é gerado e um thread tem seu sinalizador elevado na seção crítica, o outro thread não pode entrar na seção crítica ao mesmo tempo.
2. Espera Limitada. Supondo que a seção crítica termine em tempo finito, um thread, uma vez que tenha saído da seção crítica, não poderá recuperar a seção crítica. O motivo é que a variável turn é configurada para o outro segmento, o que significa que esse segmento agora tem prioridade. Isso significa que um thread não pode ser substituído infinitamente por outro thread.
3. Progresso.Se o outro segmento não estiver na seção crítica, ele simplesmente continuará com uma simples verificação. Nós não fizemos nenhuma declaração sobre se os encadeamentos são aleatoriamente parados pelo planejador do sistema. Este é um cenário idealizado em que os threads continuarão executando instruções.

Solução de Peterson

Peterson publicou seu romance e solução surpreendentemente simples em 1981 (Peterson # ref-Peterson1981MythsAT). Uma versão de seu algoritmo é mostrada abaixo, que usa uma variável compartilhada turn.
\\ Candidate #5 raise my flag turn = other_thread_id while (your flag is up and turn is other_thread_id) loop // Do Critical Section stuff lower my flag
Essa solução satisfaz Exclusão Mútua, Espera Contínua e Progresso. Se a linha 2 tiver definido a curva como 2 e estiver atualmente dentro da seção crítica. O segmento # 1 chega, *define o turno de volta para 1* e agora espera até que o segmento 2 abaixe a bandeira.

1. Exclusão mútua. Vamos tentar esboçar uma prova simples novamente. Você não entra na seção crítica até que a variável do turno seja sua ou a bandeira do outro segmento não esteja ativa. Se o sinalizador do outro segmento não estiver ativo, ele não está tentando entrar na seção crítica. Essa é a primeira ação que o thread faz e a última ação que o thread desfaz. Se a variável de turno estiver definida para esse

segmento, isso significa que o outro segmento deu o controle a esse segmento. Desde que meu sinalizador é levantado e a variável de turno está definida, o outro segmento deve aguardar no loop até que o segmento atual seja concluído.

2. Espera Limitada. Depois que um encadeamento diminui, um encadeamento aguardando no loop while sairá porque a primeira condição está quebrada. Isso significa que os segmentos não podem vencer o tempo todo.
3. Progresso.Se nenhum outro segmento estiver contestando, os sinalizadores de outros segmentos não estarão ativos. Isso significa que um segmento pode passar pelo loop while e fazer itens de seção críticos.

Extra: Posso apenas implementar a Exclusão de Software em C?

Sim - e com um pouco de pesquisa, é possível até hoje encontrá-lo em produção para processadores móveis simples específicos. O algoritmo de Peterson é usado para implementar bloqueios de kernel Linux de baixo nível para o processador móvel Tegra (um processo ARM system-on-chip e núcleo GPU da Nvidia) Link to Lock Source
No entanto, em geral, CPUs e compiladores C podem reordenar as instruções da CPU ou usar valores de cache locais específicos do núcleo da CPU que são obsoletos se outro núcleo atualizar as variáveis compartilhadas. Assim, um simples pseudo-código para a implementação de C é muito ingênuo para a maioria das plataformas. Você pode parar de ler agora.
Então, você decidiu continuar lendo. Bem, aqui estão dragões! Não diga que não avisamos. Considere este avançado e gnarly tópico mas (alerta de spoiler) um final feliz.
Considere o seguinte código,
while (flag2) { /* busy loop - go around again */
Um compilador eficiente inferiria que a flag2variável nunca é alterada dentro do loop, de modo que o teste pode ser otimizado para while(true)Using volatilegoes somewhileay para evitar otimizações do compilador desse tipo.
Instruções independentes podem ser reordenadas por um compilador otimizador ou em tempo de execução por uma otimização de execução fora de ordem pela CPU. Essas otimizações sofisticadas, se o código exigir que as variáveis sejam modificadas e verificadas, e uma ordem precisa.
Um desafio relacionado é que os núcleos da CPU incluam um cache de dados para armazenar valores de memória principal recentemente lidos ou modificados. Os valores modificados não podem ser gravados de volta na memória principal ou relidos da memória imediatamente. Assim, as alterações de dados, como o estado de uma flag e a variável turn nos exemplos acima, não podem ser compartilhadas entre dois códigos de CPU.
Mas há final feliz. O hardware moderno resolve esses problemas usando "cercas de memória", também conhecidas como barreira de memória. Isso impede que as instruções sejam encomendadas antes ou depois da barreira. Há uma perda de desempenho, mas é necessária para programas corretos!
Além disso, há instruções da CPU para garantir que a memória principal e o cache das CPUs estejam em um estado razoável e coerente. Primitivos de sincronização de nível mais alto, como pthread_mutex_locksão, irão chamar essas instruções de CPU como parte de sua implementação. Assim, na prática, a seção crítica ao redor com um bloqueio mutex e desbloquear chamadas é suficiente para ignorar esses problemas de nível inferior.

Para mais informações, sugerimos a seguinte publicação na web que discute a implementação do algoritmo de Peterson em um processo x86 e a documentação do Linux sobre barreiras de memória.

1. Cercas de memória
2. Barreiras de memória

Implementando Contando Semáforo

Agora que temos uma solução para o problema da seção crítica. Podemos razoavelmente implementar um mutex. Como podemos implementar outras primitivas de sincronização? Vamos começar com um semáforo. Para implementar um semáforo com uso eficiente da CPU, diremos que implementamos uma variável de condição. Implementar uma variável de condição de espaço O (1) usando apenas um mutex não é trivial, ou pelo menos uma variável de condição de heap O (1) não é trivial. Nós não queremos chamar malloc enquanto implementamos um primitivo, ou podemos bloquear! Verifique o final da seção para um exemplo.

- Podemos implementar um semáforo de contagem usando variáveis de condição.
- Cada semáforo precisa de uma contagem, uma variável de condição e um mutex
  typedef struct sem_t { ssize_t count ; pthread_mutex_t m ; pthread_condition_t cv ; } sem_t ;

Implementar sem_initpara inicializar a variável mutex e condição
int sem_init (sem_t * s , int pshared , int value) { if (pshared) { errno = ENOSYS /* 'Not implemented'*/ ; return - 1 ; } s -> count = value ; pthread_mutex_init (& s -> m , NULL); pthread_cond_init (& s -> cv , NULL); return 0 ; }
Nossa implementação de sem_postnecessidades para incrementar a contagem. Nós também acordaremos quaisquer threads dormindo dentro da variável de condição. Observe que nós bloqueamos e desbloqueamos o mutex para que apenas um thread possa estar dentro da seção crítica de cada vez.
sem_post (sem_t * s) { pthread_mutex_lock (& s -> m); s -> count ++ ; pthread_cond_signal (& s -> cv); /* A woken thread must acquire the lock, so it will also have to wait until we call unlock*/ pthread_mutex_unlock (& s -> m); }
Nossa implementação sem_waitpode precisar dormir se a contagem do semáforo for zero. Assim como sem_postencerramos a seção crítica usando o bloqueio, apenas um thread pode estar executando nosso código por vez. Observe se o segmento precisa esperar, então o mutex será desbloqueado, permitindo que outro segmento entre sem_poste nos desperte do nosso sono!
Observe que, mesmo que um thread seja ativado, antes de retornar pthread_cond_wait, ele deve readquirir o bloqueio, por isso ele terá que esperar até que sem_post termine.
sem_wait (sem_t * s) { pthread_mutex_lock (& s -> m); while (s -> count == 0) { pthread_cond_wait (& s -> cv , & s -> m); /*unlock mutex, wait, relock mutex*/ } s -> count -- ; pthread_mutex_unlock (& s -> m); }
Observe que estamos ligando a sem_postcada momento. Na prática, isso significa sem_postuma chamada desnecessária, pthread_cond_signalmesmo se não houver threads

em espera. Uma implementação mais eficiente só seria pthread_cond_signalnecessária quando necessário
/* Did we increment from zero to one- time to signal a thread sleeping inside sem_post */ if (s -> count == 1) /* Wake up one waiting thread!*/ pthread_cond_signal (& s -> cv);

Outras considerações de semáforo

- A implementação de semáforos reais pode incluir uma fila e problemas de agendamento para garantir justiça e prioridade. Significado, nós acordamos o maior fio de sono de maior prioridade.
- Além disso, um uso avançado sem_initpermite que os semáforos sejam compartilhados entre os processos. Nossa implementação só funciona para threads dentro do mesmo processo. Podemos consertar isso configurando a variável de condição e os atributos mutex.

Extra: Implementando CVs com Mutexes Sozinho

Implementar uma variável de condição usando apenas um mutex não é trivial. Aqui está um esboço de como poderíamos fazer isso.
typedef struct cv_node_ { pthread_mutex_t * dynamic ; int is_awoken ; struct cv_node_ * next ; } cv_node ; typedef struct { cv_node_ * head } cond_t void cond_init (cond_t * cv) { cv -> head = NULL ; cv -> dynamic = NULL ; } void cond_destroy (cond_t * cv) { // Nothing to see here // Though may be useful for the future to put pieces } static int remove_from_list (cond_t * cv , cv_node * ptr) { // Function assumes mutex is locked // Some sanity checking if (ptr == NULL) { return } // Special case head if (ptr == cv -> head) { cv -> head = cv -> head -> next ; return ; } // Otherwise find the node previous for (cv_node * prev = cv -> head ; prev -> next ; prev = prev -> next) { // If we've found it, patch it through if (prev -> next == ptr) { prev -> next = prev -> next -> next ; return ; } // Otherwise keep walking prev = prev -> next ; } // We couldn't find the node, invalid call }
Este é todo o material de definição chato. O material interessante está abaixo.
void cond_wait (cond_t * cv , pthread_mutex_t * m) { // See note (dynamic) below if (cv -> dynamic == NULL) { cv -> dynamic = m } else if (cv -> dynamic != m) { // Error can't wait with a different mutex! abort (); } // mutex is locked so we have the critical section right now // Create linked list node _on the stack_ cv_node my_node ; my_node . is_awoken = 0 ; my_node . next = cv -> head ; cv -> head = my_node . next ; pthread_mutex_unlock (m); // May do some cache busting here while (my_node == 0) { pthread_yield (); } pthread_mutex_lock (m); remove_from_list (cv , & my_node); // The dynamic binding is over if (cv -> head == NULL) { cv -> dynamic = NULL ; } } void cond_signal (cond_t * cv) { for (cv_node * iter = cv -> head ; iter ; iter = iter -> next) { // Signal makes sure one thread that has not woken up // is woken up if (iter -> is_awoken == 0) { // DON'T remove from the linked list here // There is no mutual exclusion, so we could // have a race condition iter -> is_awoken = 1 ; return ; } } // No more threads to free! No-op } void cond_broadcast (cond_t * cv) { for (cv_node * iter = cv -> head ; iter ; iter = iter -> next) { // Wake everyone up! iter -> is_awoken = 1 ; } }
Então, como isso funciona? Em vez de espaço malloc que poderia levar ao impasse. Mantemos as estruturas de dados ou os nós da lista vinculada na pilha de cada encadeamento. A lista encadeada na função de espera é criada **Enquanto o**

encadeamento tem o bloqueio de mutex, isso é importante porque, caso contrário, podemos ter uma condição de corrida na inserção e remoção. Uma implementação mais robusta teria uma variável mutex por condição.
Qual é a nota sobre (dinâmica)? Nas páginas man do pthread, wait cria uma ligação de tempo de execução para um mutex. Isso significa que, depois que a primeira chamada é chamada, um mutex é associado a uma variável de condição enquanto ainda há um encadeamento aguardando essa variável de condição. Cada novo segmento que entra deve ter o mesmo mutex e deve estar bloqueado. Portanto, o início e o fim da espera (tudo além do loop while) são mutuamente exclusivos. Após o último segmento sair, o que significa que quando head é NULL, a ligação é perdida.
As funções de sinal e transmissão apenas informam um ou todos os segmentos, respectivamente, de que devem ser acordados. **Ele não modifica as listas vinculadas porque não há mutex para impedir a corrupção se dois segmentos chamarem sinal ou difusão**
Agora um ponto avançado. Você vê como uma transmissão poderia causar um despertar espúrio neste caso? Considere esta série de eventos.

1. Alguns números mais de 2 tópicos começam a esperar
2. Outro segmento chama transmissão.
3. Essa transmissão de chamada de segmento é SIGSTOP'ed antes de virar qualquer tópicos para acordar.
4. Um outro segmento chama espera na variável de condição e se adiciona à fila.
5. Broadcast itera e libera todos os threads.

Não há garantia de *quando a* transmissão foi chamada e quando os segmentos foram adicionados em um mutex de alto desempenho. As maneiras de evitar esse comportamento são incluir carimbos de data / hora do Lamport ou exigir que a difusão seja chamada com o mutex em questão. Dessa forma, algo que *acontece - antes* da chamada de difusão não é sinalizado depois. O mesmo argumento é apresentado para o sinal também.
Você também percebeu algo mais? **É por isso que pedimos que você sinalize ou transmita antes de desbloquear** . Se você transmitir depois de desbloquear, o tempo que a transmissão leva pode ser infinito!

1. Broadcast é chamado em uma fila de espera de threads
2. O primeiro thread é liberado, o thread de transmissão está congelado. Como o mutex está desbloqueado, ele bloqueia e continua.
3. Continua por tanto tempo que chama de broadcast novamente.
4. Com nossa implementação de uma variável de condição, isso seria encerrado. Se você tivesse uma implementação que fosse anexada à parte final da lista e iterada da cabeça para a cauda, isso poderia continuar infinitamente várias vezes.

Em sistemas de alto desempenho, queremos ter certeza de que cada thread que chama wait não seja passado por outro thread que chama wait. Com a API atual que temos, não podemos garantir isso. Teríamos que pedir aos usuários que passassem um mutex ou usassem um mutex global. Em vez disso, dizemos aos programadores para sempre sinalizar ou transmitir antes de desbloquear.

Barreiras

Suponha que desejamos executar um cálculo multiencadeado que tenha dois estágios, mas não queremos avançar para o segundo estágio até que o primeiro estágio seja concluído. Poderíamos usar um método de sincronização chamado **barreira** . Quando um fio atinge uma barreira, ele irá esperar na barreira até que todos os fios atinjam a barreira, e então todos eles continuarão juntos.

Pense nisso como estar fora para uma caminhada com alguns amigos. Você faz uma anotação mental de quantos amigos você tem e concorda em esperar um pelo outro no topo de cada colina. Digamos que você seja o primeiro a chegar ao topo da primeira colina. Você vai esperar lá no topo para seus amigos. Um por um, eles chegarão ao topo, mas ninguém continuará até que a última pessoa do grupo chegue. Depois disso, todos vocês continuarão.

Pthreads tem uma função pthread_barrier_wait()que implementa isso. Você precisará declarar uma pthread_barrier_tvariável e inicializá-la com pthread_barrier_init(). pthread_barrier_init()leva o número de threads que estarão participando da barreira como um argumento. Aqui está um exemplo de programa usando barreiras.

#define _GNU_SOURCE #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <pthread.h> #include <time.h> #define THREAD_COUNT 4 pthread_barrier_t mybarrier ; void * threadFn (void * id_ptr) { int thread_id = * (int *) id_ptr ; int wait_sec = 1 + rand () % 5 ; printf ("thread %d: Wait for %d seconds. \n " , thread_id , wait_sec); sleep (wait_sec); printf ("thread %d: I'm ready... \n " , thread_id); pthread_barrier_wait (& mybarrier); printf ("thread %d: going! \n " , thread_id); return NULL ; } int main () { int i ; pthread_t ids [THREAD_COUNT]; int short_ids [THREAD_COUNT]; srand (time (NULL)); pthread_barrier_init (& mybarrier , NULL , THREAD_COUNT + 1); for (i = 0 ; i < THREAD_COUNT ; i ++) { short_ids [i] = i ; pthread_create (& ids [i], NULL , threadFn , & short_ids [i]); } printf ("main() is ready. \n "); pthread_barrier_wait (& mybarrier); printf ("main() is going! \n "); for (i = 0 ; i < THREAD_COUNT ; i ++) { pthread_join (ids [i], NULL); } pthread_barrier_destroy (& mybarrier); return 0 ; }

Agora vamos implementar nossa própria barreira e usá-la para manter todos os threads sincronizados em um grande cálculo.

double data[256][8192] 1 Threads do first calculation (use and change values in data) 2 Barrier! Wait for all threads to finish first calculation before continuing 3 Threads do second calculation (use and change values in data)

A função de rosca tem quatro partes principais

void * calc (void * arg) { /* Do my part of the first calculation */ /* Am I the last thread to finish? If so wake up all the other threads! */ /* Otherwise wait until the other threads has finished part one */ /* Do my part of the second calculation */ }

Nosso thread principal criará os 16 threads e dividiremos cada cálculo em 16 partes separadas. Cada thread receberá um valor único (0,1,2, .. 15), para poder trabalhar em seu próprio bloco. Como um tipo (void *) pode conter inteiros pequenos, passaremos o valor de ium objeto void para um ponteiro vazio.

#define N (16) double data [256][8192] ; int main () { pthread_t ids [N]; for (int i = 0 ; i < N ; i ++) pthread_create (& ids [i], NULL , calc , (void *) i);

Note, nunca iremos desreferenciar este valor de ponteiro como um local de memória real. Vamos apenas lançá-lo de volta para um inteiro.

void * calc (void * ptr) { // Thread 0 will work on rows 0..15, thread 1 on rows 16..31 int x , y , start = N * (int) ptr ; int end = start + N ; for (x = start ; x < end ; x ++) for (y = 0 ; y < 8192 ; y ++) { /* do calc #1 */ }

Depois que o cálculo 1 terminar, precisamos aguardar os threads mais lentos, a menos que estejamos o último segmento !. Então, acompanhe o número de threads que chegaram ao nosso 'checkpoint' de barreira:

// Global: int remain = N ; // After calc #1 code: remain -- ; // We finished if (remain == 0) { /*I'm last! - Time for everyone to wake up! */ } else { while (remain != 0) { /* spin spin spin*/ } }

No entanto, o código tem algumas falhas. Um é dois segmentos podem tentar diminuir remain. O outro é o loop é um loop ocupado. Nós podemos fazer melhor! Vamos usar uma variável de condição e, em seguida, usaremos as funções de transmissão / sinal para ativar os threads de sono.

Um lembrete de que uma variável de condição é semelhante a uma casa! Tópicos vão lá para dormir (pthread_cond_wait). Você pode optar por ativar um thread (pthread_cond_signal) ou todos eles (pthread_cond_broadcast). Se não houver segmentos atualmente aguardando, essas duas chamadas não terão efeito.

Uma versão de variável de condição é geralmente muito semelhante a uma solução incorreta de loop ocupado - como mostraremos a seguir. Primeiro, vamos adicionar um mutex e condicionar variáveis globais e não se esqueça de inicializá-las main.

//global variables pthread_mutex_t m ; pthread_cond_t cv ; main () { pthread_mutex_init (& m , NULL); pthread_cond_init (& cv , NULL);

Vamos usar o mutex para garantir que apenas um thread seja modificado de remaincada vez. O último segmento que chega precisa despertar *todos os* threads para dormir - então vamos usar pthread_cond_broadcast(&cv)nãopthread_cond_signal

pthread_mutex_lock (& m); remain -- ; if (remain == 0) { pthread_cond_broadcast (& cv); } else { while (remain != 0) { pthread_cond_wait (& cv , & m); } } pthread_mutex_unlock (& m);

Quando um thread entra pthread_cond_wait, ele libera o mutex e dorme. Em algum momento no futuro, será acordado. Uma vez que trazemos um fio de volta do seu sono, antes de retornar ele deve esperar até que ele possa bloquear o mutex. Observe que, mesmo que um encadeamento do sono acorde cedo, ele verificará a condição do loop while e entrará novamente em espera, se necessário.

A barreira acima não é reutilizável . O que significa que se o colocarmos em qualquer loop de cálculo antigo, há uma boa chance de que o código encontre uma condição em que a barreira se torne deadlocks ou um thread avança uma iteração mais rapidamente. Por que é que? Estou feliz que você tenha perguntado.

O conceito aqui é o que chamamos de segmento ambicioso. Vamos supor que um thread é muito mais rápido que todos os outros threads. Com a API de barreira, esse thread deve estar aguardando, mas talvez não seja. Para torná-lo concreto, vamos olhar para este código

barrier_wait (barrier * b) { pthread_mutex_lock (& b -> m); // If it is 0 before decrement, we should be on // another iteration right? if (b -> remain == 0) b -> remain = NUM_THREADS ; b -> remain -- ; if (b -> remain == 0) { pthread_cond_broadcast (& cv); } else { while (b -> remain != 0) { pthread_cond_wait (& cv , & m); } } pthread_mutex_unlock (& b -> m); } for () { // Some calc barrier_wait (b); }

O que acontece se você tiver o que chamamos de **segmento ambicioso?** . Bem

1. Muitos outros segmentos aguardam a variável de condição
2. O último segmento transmite.
3. Um único segmento deixa o loop while.
4. Este thread único executa seu cálculo antes que qualquer outro thread *ative*
5. Redefina o número de threads restantes e volta a dormir.

Agora o problema se torna todos os outros segmentos que deveriam ter acordado nunca fazer e nossos deadlocks de implementação. Como você resolveria isso? Dica: Se vários segmentos chamarem barrier_waitum loop, é possível garantir que eles estejam na mesma iteração.

Problema do gravador de leitor

Imagine que você tivesse uma estrutura de dados de mapa de valor-chave que é usada por muitos segmentos. Vários encadeamentos devem poder procurar valores (lidos) ao mesmo tempo, desde que a estrutura de dados não esteja sendo gravada. Os escritores não são tão gregários. Para evitar corrupção de dados, apenas um thread de cada vez pode modificar (write) a estrutura de dados e nenhum leitor pode estar lendo naquele momento. Este é um exemplo do *problema* do *Reader Writer* . Ou seja, como podemos sincronizar eficientemente vários leitores e escritores de modo que vários leitores possam ler juntos, mas um gravador obtém acesso exclusivo?
Uma tentativa incorreta é mostrada abaixo ("lock" é um atalho para pthread_mutex_lock):

Tentativa # 1

void read () { lock (& m) // do read stuff unlock (& m) } void write () { lock (& m) // do write stuff unlock (& m) }
Pelo menos a nossa primeira tentativa não sofre corrupção de dados. Os leitores devem esperar enquanto um escritor está escrevendo e vice-versa! No entanto, os leitores também devem esperar por outros leitores. Vamos tentar outra implementação.

Tentativa # 2:

void read () { while (writing) { /*spin*/ } reading = 1 // do read stuff reading = 0 } void write () { while (reading || writing) { /*spin*/ } writing = 1 // do write stuff writing = 0 }
Nossa segunda tentativa sofre de uma condição de corrida. Imagine se dois segmentos chamados reade writeou ambos chamados escrevessem ao mesmo tempo. Ambos os segmentos poderão prosseguir! Em segundo lugar, podemos ter vários leitores e vários escritores, portanto, vamos acompanhar o número total de leitores ou escritores que nos leva a tentar # 3.

Tentativa # 3

Lembre-se disso pthread_cond_wait executa *três* ações. Em primeiro lugar, destrava atomicamente o mutex e depois dorme (até ser acordado por pthread_cond_signalou pthread_cond_broadcast). Em terceiro lugar, o encadeamento acordado deve readquirir o

bloqueio mutex antes de retornar. Portanto, apenas um thread pode estar sendo executado dentro da seção crítica definida pelos métodos lock e unlock ().
Implementação # 3 abaixo garante que um leitor entrará no cond_wait se houver escritores escrevendo.
read () { lock (& m) while (writing) cond_wait (& cv , & m) reading ++ ; /* Read here! */ reading -- cond_signal (& cv) unlock (& m) }
No entanto, apenas um leitor por vez pode ler porque o candidato # 3 não desbloqueou o mutex. Uma versão melhor desbloqueia antes de ler.
read () { lock (& m); while (writing) cond_wait (& cv , & m) reading ++ ; unlock (& m) /* Read here! */ lock (& m) reading -- cond_signal (& cv) unlock (& m) }
Isso significa que um escritor e uma leitura poderiam ler e escrever ao mesmo tempo? Não!Em primeiro lugar, lembre-se de que cond_wait exige que o encadeamento readquira o bloqueio mutex antes de retornar. Assim, apenas um thread pode estar executando código dentro da seção crítica (marcada com **) por vez!
read () { lock (& m); ** while (writing) ** cond_wait (& cv , & m) ** reading ++ ; unlock (& m) /* Read here! */ lock (& m) ** reading -- ** cond_signal (& cv) unlock (& m) }
Os escritores devem esperar por todos. A exclusão mútua é assegurada pela fechadura.
write () { lock (& m); ** while (reading || writing) ** cond_wait (& cv , & m); ** writing ++ ; ** ** /* Write here! */ ** writing -- ; ** cond_signal (& cv); unlock (& m); }
Candidato # 3 acima também usa pthread_cond_signal . Isso só vai acordar um segmento. Por exemplo, se muitos leitores estão esperando o escritor completar, apenas um leitor adormecido será acordado de seu sono. O leitor e o gravador devem usar cond_broadcastpara que todos os segmentos sejam ativados e verifiquem sua condição de loop while.

Escritores famintos

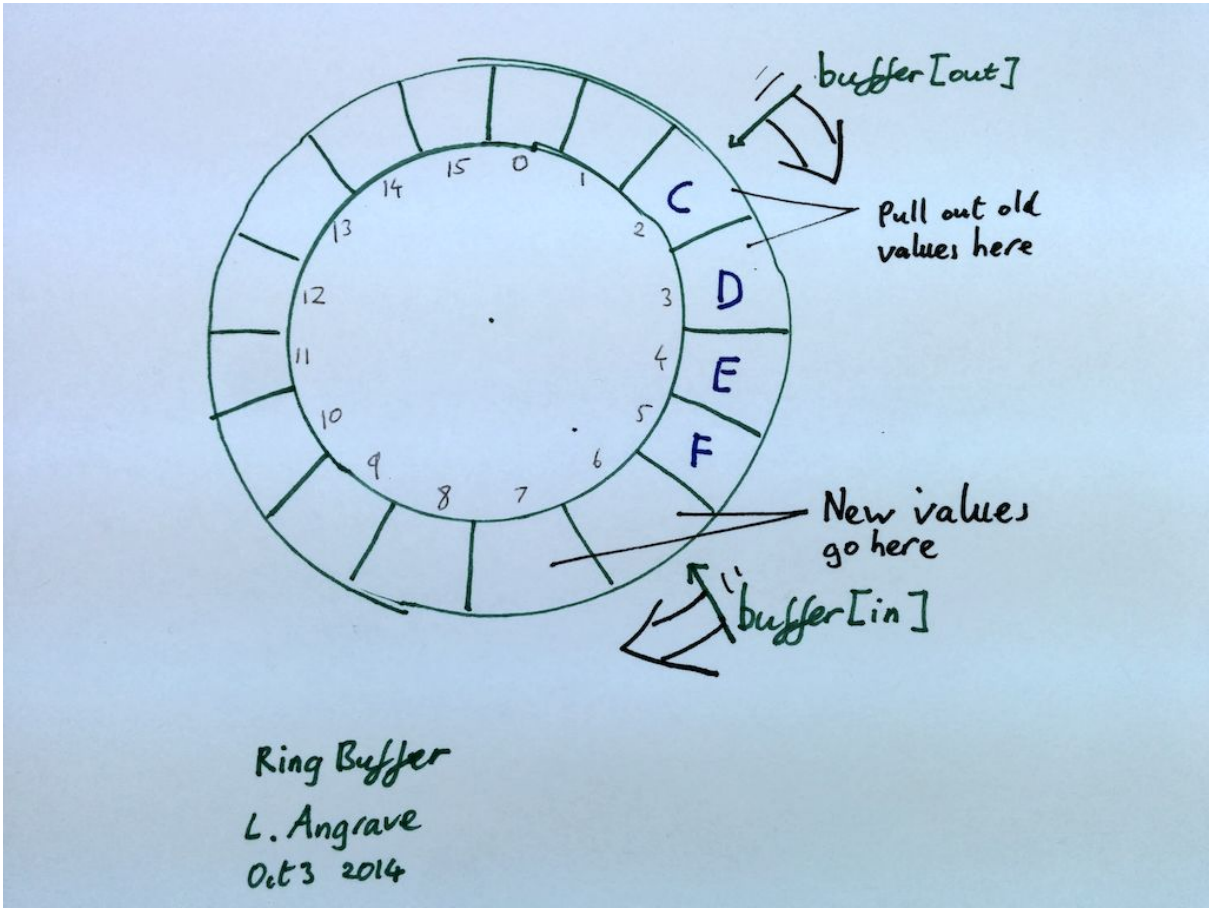
Candidato # 3 acima sofre de fome. Se os leitores estão constantemente chegando, então um escritor nunca será capaz de prosseguir (a contagem de 'leitura' nunca se reduz a zero). Isso é conhecido como *fome* e seria descoberto sob cargas pesadas. Nossa correção é implementar uma espera limitada pelo gravador. Se um escritor chegar, ele ainda precisará aguardar os leitores existentes, no entanto, os futuros leitores devem ser colocados em uma “caneta de espera” e esperar que o escritor termine. A “caneta de espera” pode ser implementada usando uma variável e uma variável de condição para que possamos acordar os threads assim que o gravador terminar.
Nosso plano é que, quando um escritor chegar, e antes de esperar que os leitores atuais terminem, registrar nossa intenção de escrever incrementando um contador 'escritor'
write () { lock () writer ++ while (reading || writing) cond_wait unlock () ... }
E os leitores entrantes não poderão continuar enquanto o gravador não for zero. Observe que 'escritor' indica que um escritor chegou, enquanto 'leitura' e 'escrita' indicam que há um leitor ou escritor *ativo* .
read () { lock () // readers that arrive *after* the writer arrived will have to wait here! while (writer) cond_wait (& cv , & m) // readers that arrive while there is an active writer // will also wait. while (writing) cond_wait (& cv , & m) reading ++ unlock ... }

Tentativa # 4

Abaixo está a nossa primeira solução de trabalho para o problema do Reader-Writer. Observe que, se você continuar a ler sobre o “problema do Reader Writer”, descobrirá que resolvemos o problema do “Second Reader Writer” dando aos escritores um acesso preferencial ao bloqueio. Esta solução não é ideal. No entanto, ele satisfaz nosso problema original de N leitores ativos, escritor único ativo, e evitando a fome do escritor, se houver um fluxo constante de leitores.
Você consegue identificar alguma melhoria? Por exemplo, como você melhoraria o código para que nós apenas despertássemos leitores ou um escritor?
int writers ; // Number writer threads that want to enter the critical section (some or all of these may be blocked) int writing ; // Number of threads that are actually writing inside the CS (can only be zero or one) int reading ; // Number of threads that are actually reading inside the CS // if writing !=0 then reading must be zero (and vice versa) reader () { lock (& m) while (writers) cond_wait (& turn , & m) // No need to wait while(writing here) because we can only exit the above loop // when writing is zero reading ++ unlock (& m) // perform reading here lock (& m) reading -- cond_broadcast (& turn) unlock (& m) } writer () { lock (& m) writers ++ while (reading || writing) cond_wait (& turn , & m) writing ++ unlock (& m) // perform writing here lock (& m) writing -- writers -- cond_broadcast (& turn) unlock (& m) }

Buffer de anel

Um buffer de anel é um mecanismo de armazenamento simples, geralmente de tamanho fixo, em que a memória contígua é tratada como se fosse circular e dois contadores de índice acompanham o início e o fim da fila atuais. Como a indexação de matriz não é circular, os contadores de índice devem ficar em torno de zero quando movidos após o final da matriz. À medida que os dados são adicionados (enfileirados) à frente da fila ou removidos (em fila de espera) da cauda da fila, os itens atuais no buffer formam um trem que parece circular na trilha.



Uma implementação simples (single-threaded) é mostrada abaixo. Observe que enfileiramento e desenfileiramento não protegem contra estouro ou estouro. É possível adicionar um item quando a fila está cheia e é possível remover um item quando a fila está vazia. Por exemplo, se adicionássemos 20 inteiros (1,2,3…) à fila e não desenredássemos nenhum item, os valores 17,18,19,20substituiriam o 1,2,3,4. Não corrigiremos esse problema agora, em vez disso, quando criarmos a versão multithread, garantiremos que os encadeamentos de enfileiramento e remoção de enfileiramento sejam bloqueados enquanto o buffer de toques estiver cheio ou vazio, respectivamente.
void * buffer [16]; int in = 0 , out = 0 ; void enqueue (void * value) { /* Add one item to the front of the queue*/ buffer [in] = value ; in ++ ; /* Advance the index for next time */ if (in == 16) in = 0 ; /* Wrap around! */ } void * dequeue () { /* Remove one item to the end of the queue.*/ void * result = buffer [out]; out ++ ; if (out == 16) out = 0 ; return result ; }

Gotchas de Buffer de Anel

É muito tentador escrever o enfileiramento ou o método de enfileiramento na seguinte forma compacta.
// N is the capacity of the buffer void enqueue (void * value) b [(in ++) % N] = value ; }
Este método parece funcionar (passar por testes simples, etc), mas contém um bug sutil. Com operações de enfileiramento suficientes (um pouco mais de dois bilhões), o valor int inirá transbordar e se tornar negativo! O operador de módulo (ou 'restante') %preserva o sinal. Assim você pode acabar escrevendo emb[-14] por exemplo!
Uma forma compacta é correta, usando o mascaramento de bits, desde que N seja 2 ^ x (16,32,64,…)

b [(in ++) & (N - 1)] = value ;
Esse buffer ainda não impede o estouro ou estouro de buffer. Para isso, vamos voltar para a nossa tentativa multi-threaded que irá bloquear um segmento até que haja espaço ou haja pelo menos um item para remover.

Correção Multithreaded

O código a seguir é uma implementação incorreta. O que vai acontecer?Vontade enqueuee / ou dequeuebloqueio? A exclusão mútua é satisfeita? Pode o buffer underflow? O buffer pode estourar? Para maior clareza pthread_mutexé encurtado p_me assumimos sem_wait não pode ser interrompido.
#define N 16 void * b [N] int in = 0 , out = 0 p_m_t lock sem_t s1 , s2 void init () { p_m_init (& lock , NULL) sem_init (& s1 , 0 , 16) sem_init (& s2 , 0 , 0) } enqueue (void * value) { p_m_lock (& lock) // Hint: Wait while zero. Decrement and return sem_wait (& s1) b [(in ++) & (N - 1)] = value // Hint: Increment. Will wake up a waiting thread sem_post (& s1) p_m_unlock (& lock) } void * dequeue (){ p_m_lock (& lock) sem_wait (& s2) void * result = b [(out ++) & (N - 1)] sem_post (& s2) p_m_unlock (& lock) return result }

Análise

Antes de ler, veja quantos erros você pode encontrar. Em seguida, determine o que aconteceria se os encadeamentos fossem chamados de métodos de enfileiramento e de enfileiramento.

- O método de enfileiramento espera e posta no mesmo semáforo (s1) e similarmente com enfileiramento e (s2) isto é, nós diminuímos o valor e imediatamente incrementamos o valor, então ao final da função o valor do semáforo é inalterado!
- O valor inicial de s1 é 16, portanto, o semáforo nunca será reduzido a zero - o enfileiramento não será bloqueado se o buffer de anel estiver cheio
    - então estouro é possível.
- O valor inicial de s2 é zero, portanto, as chamadas para desenfileiramento sempre serão bloqueadas e nunca mais retornarão!
- A ordem de bloqueio mutex e sem_wait precisará ser trocada, no entanto, este exemplo está tão quebrado que esse bug não tem efeito!

Outra análise

O código a seguir é uma implementação incorreta. O que vai acontecer?Vontade enqueuee / ou dequeuebloqueio? A exclusão mútua é satisfeita? Pode o buffer underflow? O buffer pode estourar? Para maior clareza pthread_mutexé encurtado p_me assumimos sem_wait não pode ser interrompido.
void * b [16] int in = 0 , out = 0 p_m_t lock sem_t s1 , s2 void init () { sem_init (& s1 , 0 , 16) sem_init (& s2 , 0 , 0) } enqueue (void * value){ sem_wait (& s2) p_m_lock (& lock) b [(in ++) & (N - 1)] = value p_m_unlock (& lock) sem_post (& s1) } void * dequeue (){ sem_wait (& s1) p_m_lock (& lock) void * result = b [(out ++) & (N - 1)] p_m_unlock (& lock) sem_post (& s2) return result ; }
Aqui estão alguns problemas que esperamos que você tenha encontrado.

- O valor inicial de s2 é 0. Assim, o enfileiramento irá bloquear na primeira chamada para sem_wait mesmo que o buffer esteja vazio!
- O valor inicial de s1 é 16. Assim, o dequeue não será bloqueado na primeira chamada para sem_wait mesmo que o buffer esteja vazio - Underflow! O método de desenfileiramento retornará dados inválidos.
- O código não satisfaz a Exclusão Mútua. Dois segmentos podem modificar inou outao mesmo tempo! O código parece usar o bloqueio mutex. Infelizmente, a trava nunca foi inicializada com pthread_mutex_init()ou PTHREAD_MUTEX_INITIALIZER- então a trava pode não funcionar (pthread_mutex_lockpode simplesmente não fazer nada)

Implementação correta de um buffer de anel

Como o bloqueio mutex é armazenado na memória global (estática), ele pode ser inicializado com PTHREAD_MUTEX_INITIALIZER. Se tivéssemos alocado espaço para o mutex no heap, então teríamos usadopthread_mutex_init(ptr, NULL)
#include <pthread.h> #include <semaphore.h> // N must be 2^i #define N (16) void * b [N] int in = 0 , out = 0 p_m_t lock = PTHREAD_MUTEX_INITIALIZER sem_t countsem , spacesem void init () { sem_init (& countsem , 0 , 0) sem_init (& spacesem , 0 , 16) }
O método de enfileiramento é mostrado abaixo. Certifique-se de anotar.

1. O bloqueio é mantido apenas durante a seção crítica (acesso à estrutura de dados).
2. Uma implementação completa precisaria se proteger contra retornos antecipados sem_waitdevido a sinais POSIX.

enqueue (void * value){ // wait if there is no space left: sem_wait (& spacesem) p_m_lock (& lock) b [(in ++) & (N - 1)] = value p_m_unlock (& lock) // increment the count of the number of items sem_post (& countsem) }
A dequeueimplementação é mostrada abaixo. Observe a simetria das chamadas de sincronização para enqueue. Em ambos os casos, as funções aguardam primeiro se a contagem de espaços ou a contagem de itens for zero.
void * dequeue (){ // Wait if there are no items in the buffer sem_wait (& countsem) p_m_lock (& lock) void * result = b [(out ++) & (N - 1)] p_m_unlock (& lock) // Increment the count of the number of spaces sem_post (& spacesem) return result }
Alimento para o pensamento:

- O que aconteceria se a ordem pthread_mutex_unlocke as sem_postchamadas fossem trocadas?
- O que aconteceria se a ordem sem_waite as pthread_mutex_lockchamadas fossem trocadas?

Extra: Sincronização de Processos

Você pensou que estava usando processos diferentes para não precisar sincronizar? Pense de novo!Você pode não ter condições de corrida dentro de um processo, mas e se o seu

processo precisa interagir com o sistema em torno dele? Vamos considerar um exemplo motivador

```
void write_string ( const char * data ) { int fd = open ( "my_file.txt" , O_WRONLY ); write ( fd , data , strlen ( data )); close ( fd ); } int main () { if ( ! fork ()) { write_string ( "key1: value1" ); wait ( NULL ); } else { write_string ( "key2: value2" ); } return 0 ; }
```

Se nenhuma das chamadas do sistema falhar, então devemos obter algo parecido com isto, dado que o arquivo estava vazio para começar.

```
key1: value1 key2: value2 key2: value2 key1: value1
```

Interrupção

Mas há uma nuance escondida. A maioria das chamadas do sistema pode interruptedsignificar que o sistema operacional pode interromper uma chamada do sistema em andamento porque precisa interromper o processo. Então, impedindo fork wait opene closefracassando - eles normalmente vão para a conclusão - o que acontece se writefalhar? Se a gravação falhar e nenhum byte for escrito, podemos obter algo como key1: value1ou key2: value2. Esta é a perda de dados que está incorreta, mas não corromperá o arquivo. O que acontece se a gravação for interrompida após uma gravação parcial? Nós temos todo tipo de loucura. Por exemplo,

```
key2: key1: value1
```

Solução

Você pode criar um mutex antes do fork - no entanto, o processo filho e pai não compartilharão a memória virtual e cada um terá um mutex independente do outro. Nota avançada: Existem opções avançadas usando a memória compartilhada que permite que um filho e pai compartilhem um mutex se ele é criado com as opções corretas e usa um segmento de memória compartilhada. Veja http://stackoverflow.com/questions/19172541/procs-fork-and-mutexes

Então o que deveríamos fazer? Nós devemos usar um mutex compartilhado! Considere o seguinte código.

```
pthread_mutex_t * mutex = NULL ; pthread_mutexattr_t attr ; void write_string ( const char * data ) { pthread_mutex_lock ( mutex ); int fd = open ( "my_file.txt" , O_WRONLY ); int bytes_to_write = strlen ( data ), written = 0 ; while ( written < bytes_to_write ) { written += write ( fd , data + written , bytes_to_write - written ); } close ( fd ); pthread_mutex_unlock ( mutex ); } int main () { pthread_mutexattr_init ( & attr ); pthread_mutexattr_setpshared ( & attr , PTHREAD_PROCESS_SHARED ); pmutex = mmap ( NULL , sizeof ( pthread_mutex_t ), PROT_READ | PROT_WRITE , MAP_SHARED | MAP_ANON , - 1 , 0 ); pthread_mutex_init ( pmutex , & attrmutex ); if ( ! fork ()) { write_string ( "key1: value1" ); wait ( NULL ); pthread_mutex_destroy ( pmutex ); pthread_mutexattr_destroy ( & attrmutex ); munmap (( void * ) pmutex , sizeof ( * pmutex )); } else { write_string ( "key2: value2" ); } return 0 ; }
```

O que o código faz em main é inicializar um processo mutex compartilhado usando uma parte da sharedmemória. Você descobrirá o que esta chamada mmapfaz depois - apenas assuma por enquanto que ela cria memória compartilhada entre processos. Podemos inicializar uma pthread_mutex_tparte especial da memória e usá-la como normal. Para compensar a writefalha, colocamos a writechamada dentro de um loop while que continua

gravando, desde que haja bytes para escrever. Agora, se todas as outras chamadas do sistema funcionarem, deve haver mais condições de corrida.
A maioria dos programas tenta evitar esse problema gravando arquivos separados, mas é bom saber que existem mutexes nos processos, e eles são úteis. Você pode usar todos os primitivos que você aprendeu anteriormente! Barreiras, semáforos e variáveis de condição podem ser inicializados em uma parte compartilhada da memória e usados de maneira semelhante aos seus correspondentes multithreading. Você não precisa conhecer a implementação, só precisa saber que mutexes e outras primitivas de sincronização podem ser compartilhadas entre processos e alguns dos benefícios.

- Você não precisa se preocupar com endereços de memória arbitrários se tornando candidatos a condições de corrida. Isso significa que apenas áreas em que você especificamente mmapou fora de recursos do sistema, como arquivos, estão em perigo.
- Você obtém o bom isolamento de um processo, portanto, se um processo falhar, o sistema poderá manter
- Quando você tem muitos encadeamentos, criar um processo pode facilitar o carregamento do sistema

Extra: Modelos de Ordem Superior de Sincronização

Ao usar o atomics, é necessário especificar o modelo correto de sincronização para garantir que você tenha o comportamento correto para o seu programa. Você pode ler mais sobre eles https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync Estes exemplos são adaptados a partir deles.

Sequencialmente Consistente

Sequencialmente consistente é o modelo mais simples, menos propenso a erros e mais dispendioso. Este modelo diz que qualquer mudança que aconteça, todas as mudanças antes dela serão sincronizadas entre todos os threads.
Thread 1 Thread 2 1.0 atomic_store(x, 1) 1.1 y = 10 2.1 if (atomic_load(x) == 0) 1.2 atomic_store(x, 0); 2.2 y != 10 && abort();
Nunca vai desistir. Isso ocorre porque o armazenamento acontece antes da instrução if no thread 2 e y == 1 ou o armazenamento acontece depois e x não é igual a 2.

Relaxado

Relaxado é uma simples ordem de memória que oferece mais otimizações. Isso significa que apenas uma operação específica precisa ser atômica. Pode-se ter leituras e gravações obsoletas, mas depois de ler o novo valor, ele não ficará velho.
-Thread 1- -Thread 2- atomic_store(x, 1); printf("%d\n", x) // 1 atomic_store(x, 0); printf("%d\n", x) // could be 1 or 0 printf("%d\n", x) // could be 1 or 0
Mas isso significa que cargas e armazenamentos anteriores não precisam afetar outros segmentos. No exemplo anterior, o código agora pode falhar.

Adquirir / Liberar

A maneira mais simples que eu posso explicar é que a ordem das variáveis atômicas não precisa ser consistente - o que significa que se eu atribuir varo atômico a 10, então varo atômico será 0 aqueles que não precisam se propagar e você pode ficar obsoleto lê. Variáveis não atômicas precisam ser atualizadas em todos os encadeamentos.

Consumir

Imagine o mesmo que acima, exceto que variáveis não atômicas não precisam ser atualizadas em todos os tópicos. Este modelo foi introduzido para que possa haver um modelo Adquirir / Liberar / Consumir sem misturar em Descontraído porque Consumir é semelhante a relaxar.

Fontes externas

- http://linux.die.net/man/3/pthread_mutex_lock
- http://linux.die.net/man/3/pthread_mutex_unlock
- http://linux.die.net/man/3/pthread_mutex_init
- http://linux.die.net/man/3/pthread_mutex_destroy
- http://man7.org/linux/man-pages/man3/sem_init.3.html
- http://man7.org/linux/man-pages/man3/sem_wait.3.html
- http://man7.org/linux/man-pages/man3/sem_post.3.html
- http://man7.org/linux/man-pages/man3/sem_destroy.3.html

Tópicos

- Operações atômicas
- Seção Crítica
- Problema do consumidor produtor
- Usando variáveis de condição
- Usando o semáforo contando
- Implementando uma barreira
- Implementando um buffer de anel
- Usando pthread_mutex
- Implementando o consumidor produtor
- Analisando codificados multi-threaded

Perguntas

- O que é operação atômica?
- Por que o seguinte não funciona em código paralelo
  //In the global section size_t a ; //In pthread function for (int i = 0 ; i < 100000000 ; i ++) a ++ ;
  E isso vai?
  //In the global section atomic_size_t a ; //In pthread function for (int i = 0 ; i < 100000000 ; i ++) atomic_fetch_add (a , 1);

- Quais são algumas desvantagens das operações atômicas? O que seria mais rápido: manter uma variável local ou muitas operações atômicas?
- Qual é a seção crítica?
- Uma vez que você identificou uma seção crítica, qual é a única forma de assegurar que apenas uma thread estará na seção por vez?
- Identifique a seção crítica aqui
  struct linked_list ; struct node ; void add_linked_list (linked_list * ll , void * elem){ node * packaged = new_node (elem); if (ll -> head){ ll -> head = } else { packaged -> next = ll -> head ; ll -> head = packaged ; ll -> size ++ ; } } void * pop_elem (linked_list * ll , size_t index){ if (index >= ll -> size) return NULL ; node * i , * prev ; for (i = ll -> head ; i && index ; i = i -> next , index --){ prev = i ; } //i points to the element we need to pop, prev before if (prev -> next) prev -> next = prev -> next -> next ; ll -> size -- ; void * elem = i -> elem ; destroy_node (i); return elem ; }
- Quão apertado você pode fazer a seção crítica?
- O que é um problema do consumidor produtor? Como o problema acima pode ser um problema de consumo do produtor ser usado na seção acima? Como o problema do consumidor de um produtor está relacionado a um problema de leitor?
- O que é uma variável de condição? Por que há uma vantagem em usar um em um whileloop?
- Por que esse código é perigoso?
  if (not_ready){ pthread_cond_wait (& cv , & mtx); }
- O que é um semáforo de contagem? Dê-me uma analogia a um pote de biscoitos / caixa de pizza / item de comida limitada.
- O que é uma barreira de thread?
- Use um semáforo de contagem para implementar uma barreira.
- Escreva uma fila de produtor / consumidor, que tal uma pilha de consumidores?
- Dê-me uma implementação de um bloqueio leitor-escritor com variáveis de condição, fazer uma estrutura com o que você precisa, só precisa ser capaz de suportar as seguintes funções
  void reader_lock (rw_lock_t * lck); void writer_lock (rw_lock_t * lck); void reader_unlock (rw_lock_t * lck); void writer_unlock (rw_lock_t * lck);
  A única especificação é que entre reader_locke reader_unlock, nenhum escritor pode escrever. Entre os fechamentos do escritor, apenas um escritor pode estar escrevendo por vez.
- Escreva código para implementar um consumidor produtor usando SOMENTE três semáforos de contagem. Suponha que possa haver mais de um encadeamento chamando enfileiramento e desenfileiramento. Determine o valor inicial de cada semáforo.
- Escrever código para implementar um consumidor produtor usando variáveis de condição e um mutex. Suponha que possa haver mais de um encadeamento chamando enfileiramento e desenfileiramento.
- Use CVs para implementar funções de bloqueio add (int não assinado) e subtrair (unsigned int) que nunca permitem que o valor global seja maior que 100.
- Use CVs para implementar uma barreira para 15 threads.
- Quantas das seguintes afirmações são verdadeiras?
- Pode haver vários leitores ativos

- Pode haver vários escritores ativos
- Quando há um escritor ativo, o número de leitores ativos deve ser zero
- Se houver um leitor ativo, o número de escritores ativos deve ser zero
- Um escritor deve esperar até que os leitores ativos atuais tenham terminado

Dekker, TJ e Edsgar Dijkstra. 1965. “Over de Sequentialiteit van Procesbeschrijvingen.” * Arquivo EWDijkstra: Over de Sequentialiteit van Procesbeschrijvingen (EWD 35) *. Universidade do Texas Austin. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD35.html .
Peterson, Gary L. 1981. "Mitos sobre o problema da exclusão mútua". Processo. Lett. 12: 115-16.