Operating Systems

Advanced topics on synchronization

Me

October 27, 2016

Проблемы блокирующей синхронизации

- С использованием взаимного исключения может быть связан ряд проблем:
  - конвоирование и инверсия приоритетов;
  - отсутвие "композицируемости".
  - deadlock-и;

Композиция модулей

- Один из основных инженерных принципов состоит в том, чтобы собирать сложные системы из простых компонентов:
  - вы разбиваете свой код на фукнции/классы;
  - функции и классы группируются в библиотеки;
  - из библиотек и соединяющего их кода создается приложение.
- Принцип работает до тех пор пока компоненты сравнительно "просты" и "независимы":
  - каждый из них решает конкретную задачу, и решение можно проверить;
  - собирая приложение из корректных компонет мы получаем корректное приложение (с кучей но...).

Композиция локов

```
size_t squeue_size(struct squeue *q)
{
    size_t size;

    lock(&q->lock);
    size = queue_size(&q->queue);
    unlock(&q->lock);
    return size;
}

struct node *squeue_pop(struct
    ↪ squeue *q)
{
    struct node *node;

    lock(&q->lock);
    node = queue_pop(&q->queue);
    unlock(&q->lock);
    return node;
}

int main()
{
    struct squeue queue;
    ...
    if (squeue_size(&queue))
        process(squeue_pop(&queue));
    ...
}
```

- Локи не компонуются:
  - т. е. нельзя ожидать, что пару корректных, с точки зрения многопоточности, функций можно объединить в новую корректную функцию;
  - ответсвенность за использование локов ложится на пользователя.

Deadlock

- Deadlock - состояние потока, в котором он не может прогрессировать
  - пример взаимной блокировки потоков: *Thread 0* держит блокировку *Lock 0*, а поток *Thread 1* держит блокировку *Lock 1*;
  - представим, что *Thread 0* попытается взять блокировку *Lock 1*, и одновременно *Thread 1* попытается взять *Lock 0*;
  - в deadlock-е может участвовать более 2 потоков и 2 блокировок.
- Deadlock-и может быть трудно обнаружить:
  - как и многие другие проблемы связанные с конкурентностью, они могут проявляться очень редко;
  - для воспроизведения может потребоваться учесть очень много факторов.

Средства борьбы с deadlock-ами

Страусинный алгоритм

- Самый простой способ называется "страусинный алгоритм":
  - если ПО не критично к надежности;
  - если deadlock проявляется редко;
  - то можно просто забить!
- Классический пример cost/benefit анализа:
  - поиск причин проблемы потребует много ресурсов (чем реже он проявляется тем тяжелее найти причины);
  - выгода от решения проблемы минимально - раз в неделю мользователь может перезапустить компьютер/программу;
  - явно не наш метод!

Средства борьбы с deadlock-ами

Ограниченное ожидание

- Можно ограничить ожидание при вызове lock:
  - если блокировку не удалось захватить в течении, скажем, минуты, то можно подозревать deadlock;
  - можно собрать максимум возможной информации (backtrace-ы, информация о блокировке, информация о потоке, который ее держит) и вывести сообщение.
- Отличный инженерный вариант:
  - его не трудно реализовать;
  - отсутсвие deadlock-ов не гарантируется, но если он возникнет у вас будет максимум полезной информации.

Средства борьбы с deadlock-ами

Достоверное определение deadlock-а

- Долгое ожидание не гарантирует deadlock, но deadlock можно определить достоверно:
  - построим ориентированный граф, в котором каждый узел соответствует потоку;
  - ребрам графа будут соответствовать незавершенные вызовы lock;
  - наличие циклов в таком графе означает deadlock;
  - сам цикл (потоки и блокировки) - ценная отладочная информация.
- Имея такой граф зависимостей мы можем "исправлять" deadlock-и:
  - для этого достаточно "удалить" одно из ребер в цикле;
  - можно сообщить потоку, что захват блокировки не удался;
  - это поможет только, если поток может адекватно обработать такую ситуацию.

Средства борьбы с deadlock-ами

Граф исторических зависимостей

- Как находить deadlock-и до того как они произойдут?
  - давайте определять зависимости между блокировками в виде ориентированного графа;
  - вершинами графа будут различные блокировки;
  - из блокировки A в блокировку B в графе ведет ребро, если был поток, который пытался захватить A с захваченной B;
  - мы можем запустить наше приложение и построить такой граф.
- Если в графе исторических зависимостей есть цикл, то, *вероятно*, у вас в программе есть deadlock
  - при этом не учитывается количество потоков, и какой поток какие зависимости в графе породил.

Средства борьбы с deadlock-ами

Формальная верификация/Model Checking

- Формальная верификация позволяет строго формально проверить вашу программу на соответствие определенным требованиям:
  - на самом деле не программу, а модель вашей программы;
  - модель программы может быть построена автоматически;
  - построенные автоматически модели, как правило, слишком большие и на них метод не работает;
  - требования должны задаваться математической формулой, как правило формулой в одной из темпоральных логик.
- Формальная верификация работает в критически важных системах
  - Model Checking Programs

Синхронизация без блокировок

- Возможно ли синхронизовывать потоки без блокировки?
  - в простых случаях да, очевидно, мы можем атомарно обновлять некоторые, переменные используя RMW операции, без всяких блокировок;
  - если алгоритмы будут обходиться без взаимного исключения, то и проблем связанных с блокировками не будет;
  - впрочем, возможно, будут другие.
- Что значит без блокировок? Есть три вида гарантий прогресса:
  - obstruction-free;
  - lock-free;
  - wait-free.

Obstruction freedom

- Алгоритм/структура данных/etc может называться obstruction free если выполняется следующее условие:
  - если мы останавливаем *любые* потоки кроме одного в *произвольном* месте работы этих потоков;
  - то оставшийся поток гарантированно достигнет прогресса (завершит операцию над разделяемыми данными).
- Если потоки используют взаимное исключение, то они не obstruction-free:
  - если остановить поток когда он держит блокировку, то другие потоки не смогут прогрессировать.

Lock freedom

- Алгоритм/структура данных/etc может называться lock free если выполняется следующее условие:
  - *один из потоков* гарантированно прогрессирует, независимо от состояния дургих;
  - т. е. даже если мы начнем останавливать случайные потоки в случайных местах их исполнения, один из оставшихся обязательно достигнет прогресса.
- Lock freedom сильнее obstruction freedom:
  - obstruction freedom гарантирует прогресс только если дать одному из потоков исполняться мнонпольно достаточно долго;
  - lock freedom гарантирует, что какой-то из потоков точно прогрессирует, возможно, за счет неудачи других потоков.

Wait freedom

- Алгоритм/структура данных/etc может называться wait free если выполняется следующее условие:
  - каждый поток гарантированно достигает прогресса за ограниченное количество шагов, независимо от других потоков.
- Wait freedom сильнее lock freedom:
  - lock freedom гарантирует прогресс как минимум одного потока, а wait freedom гарантирует прогресс всех потоков.

Пример: lock-free stack

```
 1  #include <stdatomic.h>
 2  #include <assert.h>
 3
 4  struct stack_node {
 5          struct stack_node *next;
 6  };
 7
 8  struct stack {
 9          struct stack_node * _Atomic top;
10  };
11
12  void stack_setup(struct stack *stack)
13  {
14          atomic_store(&stack->top, NULL);
15  }
16
17  void stack_release(struct stack *stack)
18  {
19          assert(!atomic_load(&stack->top));
20  }
```

Пример: lock-free stack

```
1  void stack_push(struct stack *stack, struct stack_node *node)
2  {
3          struct stack_node *next = atomic_load_explicit(&stack->top,
4                                                         memory_order_relaxed);
5
6          do {
7                  node->next = next;
8          } while (!atomic_compare_exchange_strong_explicit(&stack->top,
9                                                         /* expected value = */&next,
10                                                        /* new value = */node,
11                                                        memory_order_release,
12                                                        memory_order_relaxed));
13 }
```

Пример: lock-free stack

```
1  struct stack_node *stack_pop(struct stack *stack)
2  {
3          struct stack_node *top = atomic_load_explicit(&stack->top,
4                                                        memory_order_consume);
5          struct stack_node *next;
6
7          do {
8                  if (!top)
9                          break;
10                 next = top->next;
11         } while (!atomic_compare_exchange_strong_explicit(&stack->top,
12                                                           /* expected value = */&top,
13                                                           /* new value = */next,
14                                                           memory_order_consume,
15                                                           memory_order_consume));
16         return top;
17 }
```

Пример: lock-free stack

- Наш простой lock-free stack очень примечателен тем, что он содержит почти все lock-free характерные ошибки, которые можно было бы совершить:
  - использование чужой/освобожденной памяти;
  - проблема ABA.

lock-free stack: протухший указатель

```
 1  struct stack_node *stack_pop(struct stack *stack)
 2  {
 3          struct stack_node *top = atomic_load_explicit(&stack->top,
 4                                          memory_order_consume);
 5          struct stack_node *next;
 6
 7          do {
 8                  if (!top)
 9                          break;
10
11                  /* top might have been freed by this time */
12                  next = top->next;
13
14          } while (!atomic_compare_exchange_strong_explicit(&stack->top,
15                                          /* expected value = */&top,
16                                          /* new value = */next,
17                                          memory_order_consume,
18                                          memory_order_consume));
19          return top;
20  }
```

lock-free stack: проблема ABA

```
1  struct stack_node *stack_pop(struct stack *stack)
2  {
3          struct stack_node *top = atomic_load_explicit(&stack->top,
4                                          memory_order_consume);
5          struct stack_node *next;
6
7          do {
8                  if (!top)
9                          break;
10                 next = top->next;
11
12         /* CAS operation successful if stack->top is equal to top
13          * but does it mean that stack->top->next is equal to next? */
14         } while (!atomic_compare_exchange_strong_explicit(&stack->top,
15                                         /* expected value = */&top,
16                                         /* new value = */next,
17                                         memory_order_consume,
18                                         memory_order_consume));
19         return top;
20 }
```

Управление памятью в lock-free алгоритмах

- Для решения проблем работы с памятью есть несколько распространенных стратегий:
  - использовать GC, в JVM таких проблем не возникает;
  - использовать один из SMR (Safe Memory Reclamation) алгоритмов:
    - Hazard Pointers;
    - Read Copy Update.
- Существуют и другие варианты:
  - использовать lock-free счетчики ссылок - не просто и медленно;
  - использовать freelist и указатели с тегами - не практично.

Q&A