Опасность существования race condition при использовании нитей исполнения

На семинарах 7-8 мы познакомились с понятием нитей исполнения в UNIX на примере использования библиотеки pthread. Нити исполнения работают в рамках одного процесса, используют части одного и того же исполняемого кода, имеют общий доступ к ресурсам, выделенным процессу, и к данным, расположенным в процессе вне стека. При этом у каждой нити исполнения есть свой собственный стек, регистровый контекст и свой специфический системный контекст. Работа набора активностей с общими ресурсами, как говорилось на лекциях и на семинарах, требует особой аккуратности. К наборам таких активностей относятся и нити исполнения одного процесса. Аккуратность важна, поскольку набор активностей может оказаться недетерминированным, т.е. иметь race condition. Наличие race condition наблюдается, например, в программе 07-2.c, где при определенных условиях обе нити исполнения могут выдать значение a равное 1 из-за неатомарности выполнения оператора a = a + 1. Правда, это случается крайне редко. Давайте рассмотрим задачу, для которой при неаккуратном написании программы неправильные результаты будут являться не исключением, а правилом.

Написание, компиляция и прогон программы с получением недетерминированных результатов

Рассмотрим следующую учебную проблему: нам требуется к некоторой переменной, изначально равной 0, прибавить 20000000 раз значение 1 и затем вывести значение этой переменной.

Для ускорения вычислений было принято решение распараллелить выполнение программы, решающей эту проблему, на 2 нитях исполнения, каждая из которых будет прибавлять к значению общей глобальной переменной, изначально равной 0, по 1 10000000 раз, а затем главная нить будет выдавать полученный результат.

***Задача 1 (5 баллов)**:*

Напишите программу, реализующую решение проблемы с помощью 2-х нитей исполнения (включая главную), используя только те сведения, которые вы уже получили на семинарах, откомпилируйте ее и запустите несколько раз. Объясните полученные результаты.

Лишних конструкций быть не должно. Задача не принимается без проверки результатов вызова функций из библиотеки pthread. Помните, что они в отличие от привычных нам системных вызовов возвращают 0 в случае успеха, а в случае неудачи возвращают положительный код ошибки.

Попытка исправления предыдущей программы с помощью алгоритма Петерсона

По-видимому, при каждом запуске вы будете получать различные результаты, в основном, очень далекие от 20000000.

В чем здесь дело? А дело в том, что какую бы вы не брали операцию инкрементации общей переменной (a = a+1; a+=1; a++; ++a), все эти операции являются неатомарными. У вас недетерминированный набор нитей исполнения, существует race condition, а критической секцией в каждой нити исполнения является увеличение значения общей переменной на 1. Иногда две нити исполнения оказываются в своих критических секциях одновременно. Они берут из памяти старое значение общей переменной и прибавляют к этому старому значению по 1, после чего записывают полученное значение в общую переменную. В результате вместо увеличения значения общей переменной на 2, оно увеличивается только на 1. Так происходит часто, и общий результат оказывается существенно меньше 20000000.

С подобной ситуацией мы уже сталкивались при изучении вопросов работы с общей памятью (семинары 7-8) для двух процессов, и там проблему удалось решить, используя алгоритм Петерсона.

***Задача 2 (10 баллов)**:*

Модернизируйте решение задачи 1 с использованием алгоритма Петерсона, откомпилируйте и запустите ее несколько раз. Попробуйте объяснить полученные результаты.

Задача не принимается без правильно определенной критической секции. Помните, что наша цель - распараллелить вычисления!

Особенности синхронизации на многоядерных системах

Мы снова получаем различные результаты при каждом запуске, но все полученные результаты существенно ближе к 20000000, чем для предыдущей программы. Работает лучше, но все равно неправильно! Как же так, дорогая редакция! Ведь для алгоритма Петерсона было теоретически доказано, что он корректен и удовлетворяет всем требованиям к программным алгоритмам синхронизации!

Все дело вот в чем. Алгоритмы синхронизации, рассмотренные на лекциях, разрабатывались для одноядерных однопроцессорных систем в так называемой модели строгой непротиворечивости памяти. Эта модель наиболее естественна и привычна всем программистам в начале обучения. Она предполагает, что при чтении значения из некоторой ячейки памяти (т.е. из некоторой переменной), вы прочтете то значение, которое было в нее записано последним по времени перед чтением.

До появления многоядерных систем так все оно и было. И на одноядерной однопроцессорной системе Ваша программа из пункта будет работать абсолютно правильно.

При разработке многоядерных процессоров у разработчиков hardware возникли некоторые проблемы. Архитектура многоядерного процессора предполагает наличие одного или нескольких каналов связи процессора с оперативной памятью, но число каналов всегда меньше, чем количество ядер. Для сокращения числа конфликтов ядер при обращении к оперативной памяти по этим каналам в процессоре присутствует несколько уровней кэша. При этом кэш самого нижнего уровня, наиболее близкий к оперативной памяти, обычно общий для всех ядер процессора, а кэши более высоких уровней у каждого ядра — свои собственные.

Допустим, что у нас изменяется значение некоторой переменной, копия которой находится у ядра, изменяющего значение, в кэше высокого уровня. Тогда для сохранения модели строгой непротиворечивости памяти, нужно немедленно синхронизировать содержимое этой переменной во всех кэшах и оперативной памяти и одновременно объявить значение этой переменной в кэшах остальных ядер неподтвержденным, чтобы в дальнейшем остальные ядра обновили значение этой переменной в своих кэшах. Такая процедура, при ее реализации после каждой выполненной команды записи в переменную, резко снижает производительность работы процессора. Поэтому в 2005 году разработчиками hardware было принято решение ослабить модель непротиворечивости памяти для сохранения высокой производительности процессора. Изменения данных в кэше синхронизируются с остальными кэшами и оперативной памятью не сразу, а по прошествии некоторого небольшого времени, когда накопится достаточно большое количество изменений. Тогда все изменения синхронизируются сразу. Это, конечно, достаточно грубое представление о том, что действительно происходит в многоядерном процессоре.

Наличие промежутка времени между изменением значения переменной и видимостью этого изменения другими ядрами приводит к тому, что при чтении значения из переменной на некотором ядре вы можете считать не последнее записанное в нее данное, а несколько более раннее. Из-за этого старые алгоритмы синхронизации начинают работать неправильно.

На программном уровне учитывать особенности работы ядер на процессорах различной архитектуры достаточно сложно и поэтому предпочтительнее использовать различные механизмы синхронизации, так как операционные системы подстраиваются под различную архитектуру. Можно было бы использовать уже известные вам механизмы синхронизации, разработанные для взаимодействия процессов, но при работе с нитями исполнения эффективнее использовать механизмы синхронизации, разработанные именно для взаимодействия нитей. Одним из таких механизмов являются мьютексы (mutexes).

Понятие мьютекса (mutex)

В стандарте POSIX для нитей исполнения мьютекс представляет собой объект, который может находиться в одном из трех состояний: *«неинициализирован»*, *«инициализирован и свободен»* и *«инициализирован и захвачен»*. Само английское название «mutex» происходит от сокращения словосочетания «mutual exclusion» и сразу выдает основное предназначение мьютекса — использование объекта для организации взаимоисключений при работе набора нитей исполнения в критических секциях.

После инициализации мьютекса нити исполнения могут совершать над ним две операции: операцию lock для захвата мьютекса и операцию unlock для освобождения мьютекса. Если мьютекс *свободен*, то при выполнении операции lock нить переводит его в состояние *захвачен*. Если операция lock выполняется над мьютексом, захваченным какой-либо другой нитью, то нить, выполняющая операцию lock, будет переведена в состояние *ожидание* до освобождения мьютекса. Освободить мьютекс операцией unlock может только та нить, которая его захватила. Если выполняется операция unlock над мьютексом, освобождения которого ждет одна или более нитей, то ровно одна из таких нитей разблокируется и сразу захватывает мьютекс.

Вообще говоря, работа с мьютексами сильно напоминает работу с бинарными или двоичными семафорами Дейкстры. Бинарные семафоры Дейкстры отличаются от классических семафоров тем, что они могут принимать только два значения — 0 и 1. Операция P(S) для таких бинарных семафоров ничем не отличается от операции P(S) для классических семафоров, а операция V(S) выглядит так: если S==0, то S=1; иначе — ничего не делать. Поэтому часто в литературе встречается формулировка: мьютексы — это реализация бинарных семафоров Дейкстры, что категорически неверно.

При использовании бинарных семафоров, после того как процесс или нить успешно прорвались через операцию P(S), операцию V(S) может выполнить **любой процесс или нить**. При использовании мьютексов операцию unlock может выполнить **только та нить**, которая захватила мьютекс с помощью операции lock. Кроме того, говорить о каком-либо значении мьютекса абсолютно бессмысленно, поскольку мьютекс — не целая переменная, а объект, обладающий состояниями.

Инициализация мьютекса

Для использования мьютекса в программе вам потребуется описать его с помощью специального типа данных `pthread_mutex_t`, определенного в файле `<pthread.h>`, например

```
pthread_mutex_t mymutex;
```

Описанный таким образом mutex находится в состоянии *«неинициализирован»*. Для инициализации мьютекса существует два способа: использование статического инициализатора или использование специальной функции. **Внимание: повторная инициализация уже инициализированного мьютекса может привести к непредсказуемым результатам, это не регламентируется в стандарте POSIX и не обладает переносимостью!**

Для применения статического инициализатора мьютекс по стандарту POSIX должен быть описан как статический объект, т.е. его описание не должно находиться в стеке. Мы с вами будем использовать стандартный статический инициализатор `PTHREAD_MUTEX_INITIALIZER`. Пример:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Другой способ инициализации сводится к использованию функции `pthread_mutex_init()`. Пример:

```
pthread_mutex_t mymutex;
int err;
err = pthread_mutex_init(&mymtex, NULL);
```

В качестве второго параметра этой функции в нашем курсе мы всегда будем использовать параметр `NULL` — инициализацию по умолчанию для стандартного мьютекса. Существуют, конечно, и другие варианты значения второго параметра функции `pthread_mutex_init()`, и другие статические инициализаторы, но они выходят за рамки нашего курса.

После инициализации мьютекс оказывается в состоянии *«инициализирован и свободен»*.

Захват и освобождение мьютекса

Здесь и далее речь идет о стандартных мьютексах.

Для захвата мьютекса нить исполнения использует функцию `pthread_mutex_lock()`. Если до вызова этой функции мьютекс находился в состоянии *«инициализирован и свободен»*, то функция переведет его в состояние *«инициализирован и захвачен»*, и нить продолжит свое выполнение. Если нить исполнения попытается захватить уже захваченный мьютекс, то она будет переведена в состояние *ожидание* до освобождения мьютекса. Если одна или несколько нитей исполнения ожидают освобождения мьютекса, то после его освобождения разблокирована будет ровно одна нить, после чего она немедленно захватит мьютекс (разблокировка нити и последующий захват мьютекса происходят атомарно).

Важно! Функция `pthread_mutex_lock()` для стандартного мьютекса не детектирует попытку повторного захвата его той же нитью, которая уже раньше его захватила. Такая попытка приведет к возникновению тупиковой ситуации и зависанию программы.

Для освобождения мьютекса, находящегося в состоянии *«инициализирован и захвачен»* используется функция `pthread_mutex_unlock()`. Освободить мьютекс может только та нить, которая ранее его захватила. Поведение функции `pthread_mutex_unlock()` для освобождения мьютекса, находящегося в состоянии *«инициализирован и свободен»*, по стандарту POSIX не определено и зависит от реализации.

Деинициализация (разрушение) мьютекса

Если программа завершила работать с мьютексом (или его не предполагается использовать длительное время), то для освобождения ресурсов процесса и операционной системы его необходимо деинициализировать (разрушить). Это выполняется с помощью функции `pthread_mutex_destroy()`. После вызова этой функции мьютекс переходит в состояние *«неинициализирован»* и при необходимости его можно повторно инициализировать.

Важно! Нельзя деинициализировать захваченный мьютекс. В этом случае функция `pthread_mutex_destroy()` вернет ошибку.

Поведение функции `pthread_mutex_destroy()` для разрушения неинициализированного мьютекса по стандарту POSIX не определено и зависит от реализации.

Общая схема организации взаимоисключений с помощью мьютекса

Теперь понятно, как организовать пролог и эпилог критических секций в наборе нитей исполнения с помощью мьютекса. Пусть у нас в наборе n нитей исполнения. Общая схема использования мьютекса выглядит так:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

//Нить исполнения i (i=0, 1, …, n-1)

while(some condition) {
    pthread_mutex_lock(&m);
    критическая секция
    pthread_mutex_unlock(&m);
}
```

Задачи на семинар

***Задача 1 (5 баллов)**:*

См. выше

***Задача 2 (10 баллов)**:*

См. выше

***Задача 3 (10 баллов)**:*

Исправьте решение задачи 2 с двумя нитями исполнения для корректной работы с использованием мьютекса.

Задача не принимается без проверки результатов вызова функций работы с мьютексами. Они, как и другие функции pthread, возвращают 0 в случае успеха, а в случае неудачи возвращают положительный код ошибки.