Лабораторная работа №1

Многопоточность. Синхронизация потоков

Многопоточность и поток

Многопоточность - свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся "параллельно", то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

По-настоящему параллельное выполнение задач возможно только в многопроцессорной системе, поскольку только в них присутствуют несколько системных конвейеров для исполнения команд. В однопроцессорной многозадачной системе поддерживается так называемое псевдопараллельное исполнение, при котором создается видимость параллельной работы нескольких процессов. В таких системах, однако, процессы выполняются последовательно, занимая малые кванты процессорного времени.

Поток - это просто контейнер, в котором находятся:

- счетчик команд
- регистры
- стек

Поток легче, чем процесс, и создание потока стоит дешевле. Потоки используют адресное пространство процесса, которому они принадлежат, поэтому потоки внутри одного процесса могут обмениваться данными и взаимодействовать с другими потоками.

В случае, когда одна программа выполняет множество задач, поддержка множества потоков внутри одного процесса позволяет:

- разделить ответственность за разные задачи между разными потоками
- повысить быстродействие

Выбор текущего потока из нескольких активных потоков, пытающихся получить доступ к процессору называется планированием. Процедура планирования обычно связана с весьма затратной процедурой диспетчеризации - переключением процессора на новый поток, поэтому планировщик должен заботиться об эффективном использовании процессора.

Поток может находиться в одном из трёх состояний:

- выполняемый - поток, который выполняется в текущий момент на процессоре
- готовый - поток ждет получения кванта времени и готов выполнять назначенные ему инструкции. Планировщик выбирает следующий поток для выполнения только из готовых потоков
- ожидающий - работа потока заблокирована в ожидании блокирующей операции.

В реальных задачах важность работы разных потоков может сильно различаться. Для контроля этого процесса был придуман приоритет работы. У каждого потока есть такое числовое значение приоритета. Если есть несколько спящих потоков, которые нужно запустить, то ОС сначала запустит поток с более высоким приоритетом. Потоки с низким приоритетом не будут простаивать, просто они будут получать меньше времени, чем другие, но выполняться все равно будут. Потоки с одинаковыми приоритетами запускаются в порядке очереди. Приоритет потока может меняться в процессе выполнения.

Чтобы защитить жизненно важные системные данные от доступа и (или) внесения изменений со стороны пользовательских приложений, в *Windows* и *Linux* используются два процессорных режима доступа (даже если процессор поддерживает более двух режимов): **пользовательский режим** и **режим ядра**.

Код пользовательского приложения запускается в пользовательском режиме, а код операционной системы (например, системные службы и драйверы устройств) запускается в режиме ядра.

Также следует отметить, что в случае выполнения системного вызова потоком и перехода из режима пользователя в режим ядра, происходит смена стека потока на стек ядра. При переключении выполнения потока одного процесса на поток другого ОС обновляет некоторые регистры процессора, которые ответственны за механизмы виртуальной памяти, так как разные процессы имеют разное виртуальное адресное пространство.

Старайтесь не злоупотреблять средствами синхронизации, которые требуют системных вызовов ядра (например мьютексы). Переключение в режим ядра - дорогостоящая операция!

Задачи многопоточности

- многопоточность широко используется в приложениях с пользовательским интерфейсом. В этом случае за работу интерфейса отвечает один поток, а какие-либо вычисления выполняются в других потоках. Это позволяет пользовательскому интерфейсу не подвисать, когда приложение занято другими вычислениями
- многие алгоритмы легко разбиваются на независимые подзадачи, которые можно выполнять в разных потоках для повышения производительности. Например, при

фильтрации изображения разные потоки могут заниматься фильтрацией разных частей изображения
- если некоторые части приложения вынуждены ждать ответа от сервера/пользователя/устройства, то эти операции можно выделить в отдельный поток, чтобы в основном потоке можно было продолжать работу, пока другой поток ждёт ответа
- и т.д.

Проблемы многопоточности
Основной проблемой при работе с потоками является то, что несколько потоков могут одновременно обращаться к общим ресурсам, что может приводить к следующим ситуациям:

- состояние гонки (race condition) - ситуация, когда результат работы программы зависит от порядка выполнения потоков. Если потоки одновременно модифицируют одни и те же данные без синхронизации, то это может привести к непредсказуемым результатам.
- взаимная блокировка (deadlock) - это ситуация, при которой два или более потоков блокируют друг друга, ожидая освобождения ресурсов, которые заняты другим потоком.

Для обеспечения корректного взаимодействия потоков существует множество методов синхронизации.

Методы синхронизации потоков

Атомарные операции
Атомарная операция - это операция, которую невозможно наблюдать в промежуточном состоянии. Она либо выполнена, либо нет. На уровне процессора атомарность обеспечивается специальными инструкциями, такими как `compare-and-swap (CAS)`, `test-and-set` и другие. Эти инструкции могут заблокировать доступ к определенному участку памяти во время выполнения операции.
В C++ атомарные операции реализуются с помощью библиотеки atomic.

Критические секции
Критическая секция - это фрагмент кода, который должен выполняться в исключительном режиме - никакие другие потоки и процессы не должны выполнять этот же фрагмент в то же время. Для управления критическими секциями в основном используются мьютексы. Принцип работы мьютекса заключается в следующем:

- поток, который хочет получить доступ к ресурсу, сначала должен захватить мьютекс
- если мьютекс уже захвачен другим потоком, поток будет ждать, пока мьютекс не станет доступен

- после завершения работы с ресурсом поток освобождает мьютекс, чтобы другие поотоки могли захватить его и получить доступ к ресурсу
  В C++ мьютексы реализуются с помощью библиотеки mutex.

Семафоры
Семафор представляет собой счетчик, который считается свободным, если значение счетчика больше нуля, и занятым при нулевом значении. В отличие от мьютекса, который допускает доступ только одного потока к ресурсу, семафор позволяет контролировать доступ к ресурсу для нескольких потоков или процессов одновременно, в зависимости от установленного счётчика. Принцип работы семафора заключается в следующем:

- семафор имеет счётчик, который инициализируется числом, равным количеству доступных ресурсов
- каждый раз, когда поток запрашивает доступ к ресурсу, значение счётчика уменьшается
  - если счётчик больше нуля, поток получает доступ к ресурсу
  - если счётчик равен нулю, поток блокируется, ожидая, пока другой поток освободит ресурс
- когда поток завершает работу с ресурсом, значение счётчика увеличивается, позволяя другим потокам получить доступ
  В C++ семафоры реализуются с помощью библиотеки semaphore.

События
Обычно событие - некоторый объект, который может находится в одном из двух состояний: занятом или свободном. С помощью событий один поток может уведомить другие потоки о том, что определённое условие или событие произошло, после чего эти потоки могут продолжить свою работу. Алгоритм синхронизации потоков на основе событий следующий:

- создается объект события
- потоки, которые необходимо синхронизировать, блокируются и ожидают, пока событие не станет сигнальным
- когда другой поток завершает определенную задачу или достигает состояния, при котором другие потоки могут продолжить выполнение, он переводит событие в сигнальное состояние
- все потоки, которые ожидали сигнала, разблокируются и продолжают выполнение
- после того, как событие было сигнализировано, оно может быть сброшено в несигнальное состояние

В C++ для реализации синхронизации потоков на основе событий используется примитив синхронизации "условная переменная" в сочетании с мьютексом (condition_variable). Для этого поток, который управляет сигналом события, должен:

- захватить мьютекс
- выполнить работу
- вызвать метод `notify_one` или `notify_all` у объекта `condition_variable`

Любой поток, который должен выполнить работу после наступления события, должен:

- захватить мьютекс
- вызвать метод `wait`, `wait_for` или `wait_until` (операции ожидания освобождают мьютекс и приостанавливают выполнение потока)
- когда получено уведомление, истёк тайм-аут или произошло ложное пробуждение, поток пробуждается, и мьютекс повторно блокируется. Затем поток должен проверить, что условие, действительно, выполнено, и возобновить ожидание, если пробуждение было ложным

Примеры кода на C++

Создание потока:

```cpp
#include <iostream>
#include <thread>

/* Функция, выполняемая в потоке */
void threadFunction(int value) {
    std::this_thread::sleep_for(std::chrono::seconds(3));
    std::cout << "This is another thread" << std::endl;
}

int main() {
    // создаем новый поток и передаем входной параметр (int value)
    // (он запускается автоматически)
    std::thread worker(threadFunction, 10);

    std::cout << "And this is the main thread" << std::endl;

    // приостанавливаем основной поток до тех пор
    // пока поток worker не закончит работу
    worker.join();

    return 0;
}
```

Атомарные операции:

```cpp
#include <iostream>
#include <thread>
```

```cpp
#include <atomic>

// атомарная переменная
std::atomic<int> shared_var(5);

/* Функция, выполняемая в потоке */
void threadFunction(int iterations) {
    for (int i = 0; i < iterations; ++i) {
        shared_var.fetch_add(1);
    }
}

int main() {
    std::cout << shared_var << std::endl;

        int thread_1_input = 100000;
        int thread_2_input = 35000;
    std::thread thread_1(threadFunction, thread_1_input);
    std::thread thread_2(threadFunction, thread_2_input);

    thread_1.join();
    thread_2.join();

    std::cout << shared_var << std::endl;

    return 0;
}
```

Мьютекс:

```cpp
#include <iostream>
#include <thread>
#include <mutex>

// мьютекс
std::mutex m_obj;

/* Функция, выполняемая в потоке */
void threadFunction() {
    std::lock_guard<std::mutex> m_lock(m_obj);
    std::this_thread::sleep_for(std::chrono::seconds(3));
    std::cout << std::this_thread::get_id() << std::endl;
}

int main() {
    std::thread thread_1(threadFunction);
    std::thread thread_2(threadFunction);

    thread_1.join();
```

```
    thread_2.join();

    return 0;
}
```

Семафор:

```
#include <iostream>
#include <thread>
#include <semaphore>

// бинарный семафор
std::binary_semaphore semaphore(1);

/* Функция, выполняемая в потоке */
void threadFunction() {
    semaphore.acquire(); // захват семафора
    std::this_thread::sleep_for(std::chrono::seconds(3));
    std::cout << std::this_thread::get_id() << std::endl;
    semaphore.release(); // освобождение семафора
}

int main() {
    std::thread thread_1(threadFunction);
    std::thread thread_2(threadFunction);

    thread_1.join();
    thread_2.join();

    return 0;
}
```

События:

```
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>

std::condition_variable cv;
std::mutex m_obj;
bool event_signal = false;

void threadFunction() {
    std::unique_lock<std::mutex> lock(m_obj);

    // ожидание сигнала
    cv.wait(lock, [] { return event_signal; });
```

```
    std::cout << "Thread has been notified!" << std::endl;
}

int main() {
    std::thread worker(threadFunction);

    m_obj.lock();
    std::cout << "Sleeping for 3 sec and notify" << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(3));

    event_signal = true;
    // сигнализируем одному ожидающему потоку
    cv.notify_one();

    m_obj.unlock();

    worker.join();
    return 0;
}
```

Задание

Разработать однопоточное и многопоточное приложение на ЯП C++, вычисляющее с заданной точностью объем или площадь поверхности фигуры вращения относительно оси *Ox* на заданном отрезке.
Исходные данные:

- отрезок [A; B]
- точность вычислений
- количество используемых потоков (от 1 до 10)

Для синхронизации потоков использовать указанные средства синхронизации, для вычисления площади поверхности или объема использовать указанный метод согласно варианта задания.

Запустить приложение несколько раз в разных конфигурациях и построить графики зависимости:

- времени решения от количества потоков для точности `0.00001`
- времени решения от точности для линейного и трехпоточного решения

Графики полученных зависимостей можно построить в любой среде.

Вариант	f(x)	Что вычислять	Средство синхронизации потоков	Метод вычисления
1	$f(x)=e^{-x}\sin x$	площадь	Атомарные операции	Центральные прямоугольники
2	$f(x)=e^{-x}\cos x$	объём	Мьютексы	Левые прямоугольники
3	$f(x)=e^{\cos x}\sin x$	площадь	Семафоры	Правые прямоугольники
4	$f(x)=e^{\sin x}\cos x$	объём	Атомарные операции	Метод Симпсона
5	$f(x)=e^{x}arctgx$	площадь	Семафоры	Метод Монте-Карло
6	$f(x)=e^{-x}arctgx$	объём	Атомарные операции	Метод 3/8 Симпсона
7	$f(x)=x^{3}\cos x$	площадь	Критические секции	Двухточечная квадратура Гаусса-Лежандра
8	$f(x)=x^{3}\sin x$	объём	Семафоры	Трёхточечная квадратура Гаусса-Лежандра
9	$f(x)=x^{3}2^{\sqrt[3]{x\sin x\cos x}}$	площадь	Мьютексы	Центральные прямоугольники
10	$f(x)=x^{2}2^{\sqrt[3]{x\sin x\cos x}}$	объём	Атомарные операции	Левые прямоугольники
11	$f(x)=x^{2}2^{\sqrt[3]{x^{2}\sin x\cos x}}$	площадь	Атомарные операции	Правые прямоугольники
12	$f(x)=x^{2}e^{\cos x}$	объём	Мьютексы	Метод Симпсона
13	$f(x)=x^{3}e^{\sin x}$	площадь	Мьютексы	Метод Монте-Карло
14	$f(x)=x^{3}e^{\cos x}$	объём	Мьютексы	Метод 3/8 Симпсона
15	$f(x)=x^{2}e^{\sin x}$	площадь	События	Двухточечная квадратура Гаусса-Лежандра
16	$f(x)=2^{x}e^{\sin x}$	объём	Атомарные операции	Трёхточечная квадратура Гаусса-Лежандра
17	$f(x)=2^{x\sqrt{e^{\sin x}}}$	площадь	События	Центральные прямоугольники
18	$f(x)=2^{x\sqrt{e^{\sin x}}}$	объём	Семафоры	Левые прямоугольники
19	$f(x)=2^{x\sqrt{e^{\sin x\cos x}}}$	площадь	События	Правые прямоугольники
20	$f(x)=x^{2}e^{\sin x\cos x}$	объём	Критические секции	Метод Симпсона
21	$f(x)=x^{3}e^{\sin x\cos x}$	площадь	Атомарные операции	Метод Монте-Карло
22	$f(x)=x^{e^{\sin x\cos x}}$	объём	Мьютексы	Метод 3/8 Симпсона
23	$f(x)=\frac{e^{x\sin x}}{x^{2}+1}$	площадь	Семафоры	Двухточечная квадратура Гаусса-Лежандра
24	$f(x)=2^{x}e^{\cos x}$	площадь	Мьютексы	Трёхточечная квадратура Гаусса-Лежандра
25	$f(x)=x^{3}2^{\sqrt[3]{x\sin x}}$	объём	Мьютексы	Центральные прямоугольники
26	$f(x)=x^{2}2^{\sqrt[3]{x\sin x}}$	объём	Атомарные операции	Левые прямоугольники
27	$f(x)=x^{3}2^{\sqrt[3]{x\cos x}}$	площадь	Критические секции	Правые прямоугольники
28	$f(x)=x^{3}2^{\sqrt[3]{x\sin x}}$	объём	Семафоры	Метод Симпсона
29	$f(x)=\frac{2^{x\cos x\sin x}}{\sin^{2}x+\cos^{2}x}$	площадь	События	Метод Монте-Карло
30	$f(x)=\frac{e^{x\cos x}}{x^{2}+1}$	объём	Атомарные операции	Метод 3/8 Симпсона