Многопоточное программирование

Для демонстрации параллельной работы потоков ниже приведена программа, в которой два потока спорят на предмет философского вопроса «что было раньше, яйцо или курица?». Главный поток уверен, что первой была курица, о чем он и будет сообщать каждую секунду. Второй же поток раз в секунду будет опровергать своего оппонента. Всего спор продлится 5 секунд. Победит тот поток, который последним изречет свой ответ на этот, без сомнения, животрепещущий философский вопрос.

Пример на Java

class EggVoice extends Thread {

@Override

public void run() {

for(int i = 0; i < 5; i++) {

try{

sleep(1000); //Приостанавливает поток на 1 секунду

}catch(InterruptedException e){}

System.out.println("яйцо!");

} //Слово «яйцо» сказано 5 раз

}

}

public class ChickenVoice //Класс с методом main()

{

static EggVoice mAnotherOpinion; //Побочный поток

public static void main(String[] args)

{

mAnotherOpinion = new EggVoice(); //Создание потока

System.out.println("Спор начат...");

mAnotherOpinion.start(); //Запуск потока

for(int i = 0; i < 5; i++) {

try{

Thread.sleep(1000); //Приостанавливает поток на 1 секунду }catch(InterruptedException e){}

System.out.println("курица!");

} //Слово «курица» сказано 5 раз

if(mAnotherOpinion.isAlive()) //Если оппонент еще не сказал последнее слово

{

try{

mAnotherOpinion.join(); //Подождать пока оппонент закончит высказываться.

}catch(InterruptedException e){}

System.out.println("Первым появилось яйцо!");

}

else //если оппонент уже закончил высказываться

{

System.out.println("Первой появилась курица!");

}

System.out.println("Спор закончен!");

}

}

Консоль:

Спор начат...

курица!

яйцо!

яйцо!

курица!

яйцо!

курица!

яйцо!

курица!

яйцо!

курица!

Первой появилась курица!

Спор закончен!

В приведенном примере два потока параллельно в течении 5 секунд выводят информацию на консоль. Точно предсказать, какой поток закончит высказываться последним, невозможно. Можно попытаться, и можно даже угадать, но есть большая вероятность того, что та же программа при следующем запуске будет иметь другого «победителя». Это происходит из-за так называемого «асинхронного выполнения кода». Асинхронность означает то, что нельзя утверждать, что какая-либо инструкция одного потока, выполнится раньше или позже инструкции другого. Или, другими словами, параллельные потоки независимы друг от друга, за исключением тех случаев, когда программист сам описывает зависимости между потоками с помощью предусмотренных для этого средств языка.

## Состояние гонки

Состоя́ние го́нки (англ. race condition) — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

Пример:

int x;

// Поток 1:

while (!stop)

{

x++;

…

}

// Поток 2:

while (!stop)

{

if (x%2 == 0)

System.out.println("x=" + x);

…

}

Пусть x=0. Предположим, что выполнение программы происходит в таком порядке:

1. if в потоке 2 проверяет x на чётность.
2. Оператор «x++» в потоке 1 увеличивает x на единицу.
3. Оператор вывода в потоке 2 выводит «x=1», хотя, казалось бы, переменная проверена на чётность.

## Способы решения

Локальная копия

Самый простой способ решения — копирование переменной x в локальную переменную. Вот исправленный код:

// Поток 2:

while (!stop)

{

int cached\_x = x;

if (cached\_x%2 == 0)

System.out.println("x=" + cached\_x);

…

}

Естественно, этот способ работает только тогда, когда переменная одна и копирование производится за одну машинную команду.

Синхронизация

Более сложный, но и более универсальный метод решения — синхронизация потоков, а именно:

int x;

// Поток 1:

while (!stop)

{

synchronized(SomeObject)

{

x++;

}

…

}

// Поток 2:

while (!stop)

{

synchronized(SomeObject)

{

if (x%2 == 0)

System.out.println("x=" + x);

}

…

}

Комбинированный способ

Предположим, что переменная x имеет тип не int, а long (на 32-битных ЭВМ её копирование выполняется за две машинных команды), а во втором потоке вместоSystem.out.println стоит более сложная обработка. В этом случае оба метода неудовлетворительны: первый — потому что x может измениться, пока идет копирование; второй — потому что засинхронизирован слишком большой объём кода.

Эти способы можно скомбинировать, копируя «опасные» переменные в синхронизированном блоке. С одной стороны, это снимет ограничение на одну машинную команду, с другой — позволит избавиться от слишком больших синхроблоков.

long x;

// Поток 1:

while (!stop)

{

synchronized(SomeObject)

{

x++;

}

…

}

// Поток 2:

while (!stop)

{

long cached\_x;

synchronized (SomeObject)

{

cached\_x = x;

}

if (cached\_x%2 == 0)

//System.out.println("x=" + cached\_x);

DoSomethingComplicated(cached\_x);

…

}

Очевидных способов выявления и исправления состояний гонки не существует. Лучший способ избавиться от гонок — правильное проектирование многозадачной системы.

## Синхронизация потоков

Добавить новый поток в программу несложно — намного сложнее организовать его выполнение и завершение, поэтому многие функции многопоточных API предназначены именно для синхронизации потоков. В этом разделе мы кратко рассмотрим такие средства синхронизации.

Потоки координируются с помощью событий (events), которые передают информацию о состоянии одного или нескольких потоков. Событие может быть установленным (signaled) или сброшенным(unsignaled). Конкретный смысл событий может быть разным, но обычно они сигнализируют о блокировке потока.

Блокировку потока можно представить себе в виде цикла, непрерывно опрашивающего некоторую логическую переменную. Цикл продолжается до тех пор, пока переменная не примет значение TRUE. С технической точки зрения это не совсем точно, потому что заблокированный поток не производит активных опросов события. Вместо этого он приостанавливается, а система удаляет его из списка активных потоков. Лишь после того как блокирующее событие перейдет в установленное состояние, выполнение потока возобновляется. Соответственно заблокированный поток почти не расходует процессорного времени.

Пример на С++(Добавить несколько потоков в приложение, написанное с использованием библиотеки Qt, очень просто. Для этого нужно лишь создать дочерний класс от QThread и перекрыть метод run().):

class Thread : public QThread //унаследуемся от QThread

{

public:

Thread();

void setMessage(const QString &message);

void run(); //побочный поток, кроме него еще имеется основной поток - в теле главной функции

void stop();

private:

QString messageStr;

volatile bool stopped;

};

Thread::Thread()

{

stopped = false;

}

void Thread::run()

{

while (!stopped) //цикл пока не остановили поток

cerr << messageStr.ascii();

stopped = false;

cerr << endl;

}

void Thread::stop() //останавливаем поток

{

stopped = true;

}

Блокировка потоков чаще всего используется для защиты совместных ресурсов от одновременного доступа со стороны нескольких потоков.

 Мьютекс (mutex, сокращение от mutually exclusive, то есть «взаимоисключающий») представляет собой объект, который может в любой момент времени принадлежать лишь одному потоку, гарантируя безопасность доступа к связанному с ним ресурсу. Когда мьютекс принадлежит некоторому потоку, все остальные потоки, пытающиеся получить его в свое распоряжение, блокируются до освобождения мьютекса.

Пример на С++(Qt):

QMutex mutex;

void Thread::stop()

{

mutex.lock();

stopped = true; //значение переменной может изменить только1 поток одновременно

mutex.unlock();

}

Критические секции (critical section), как и мьютексы, используются для предотвращения одновременного доступа к ресурсу со стороны нескольких потоков. Однако если мьютекс может синхронизировать межпроцессные потоки, критическая секция ограничивается потоками одного процесса. Ограничение компенсируется скоростью — критическая секция работает быстрее, чем мьютекс.

Семафоры (semaphore) тоже могут применяться для ограничения доступа к ресурсам, но в отличие от мьютексов или критических секций семафор разрешает одновременный доступ со стороны нескольких потоков. Максимальное количество потоков, одновременно получающих доступ к ресурсу, определяется при создании семафора. Затем доступ предоставляется всем потокам до тех пор, пока их количество не достигнет заданного предела. Все остальные потоки, желающие получить доступ, блокируются до тех пор, пока один или несколько потоков не прекратят работу с ресурсом.

Семафор — это объект, с которым можно выполнить три операции.

init(n):

счётчик := n

enter():

ждать пока счётчик станет больше 0; после этого уменьшить счётчик на единицу.

leave():

увеличить счётчик на единицу.

Предположим, что есть такой участок кода:

semaphore.init(5);

.....

.....

void DoSomething( void )

{

semaphore.enter();

.......

semaphore.leave();

}

Тогда не более пяти потоков могут одновременно выполнять функцию DoSomething().