Threads with C++ and Qt

Version 1.0 / February 19, 2018

Anton Hofmann
<anton.hofmann@htl-salzburg.ac.at>

Contents

1 Threads with C++

Qt includes among others **class QThread**. To make code run in a separate thread, is

1. to subclass **QThread** and
2. reimplement **run()**.

For example:

```
class WorkerThread : public QThread {
  Q_OBJECT
  void run() override {
    QString result;
    /* ... here is the expensive or blocking operation ... */
    emit resultReady(result);
  }
  signals:
  void resultReady(const QString &s);
};

void MyObject::startWorkInAThread(){
  WorkerThread *workerThread = new WorkerThread(this);
  connect(workerThread, &WorkerThread::resultReady, this, &MyObject::
    handleResults);
  connect(workerThread, &WorkerThread::finished, workerThread, &QObject::
    deleteLater);
  workerThread->start();
}
```

Listing 1: QThread example

1.1 Introduction: Multithreading and Synchronization

We will learn about:

1. Lost-Update-Problem
2. Semaphore (Sperr- und Ereignissynchronisation)
3. Producer/Consumer-Problem

4. Threadsafe FIFO (bounded buffer)
5. QThread, QMutex, QWaitcondition

1.1.1 Lab-QThread: Punkt-Strich

1. create Qt console app: **OS00-PunktStrich**
2. get this code running! see http://doc.qt.io/qt-5/qthread.html

```
/**
 * Punkte und Striche werden abwechselnd gezeigt.
 * Der Scheduler ist für das Wechseln zuständig.
 */

#include <QThread>
#include <iostream>
using namespace std;

//                      !!!!!!!!!!!!!!!
class PunktStrichThread : public  QThread {
private:
  string s;
public:
  PunktStrichThread(string s){
    this->s= s;
  }
  // ***** ENTER CODE HERE ***** //
};

int main(int argc, char *argv[]){
  PunktStrichThread punkt(".");
  PunktStrichThread strich("-");

  punkt.start();
  strich.start();

  punkt.wait(5000); // wait 5 sec
  punkt.terminate();
  strich.terminate();

  return 0;
}
```

Listing 2: OS00-PunktStrich.cpp

1.1.2 Lab-QThread: Lost-Update

see https://de.wikipedia.org/wiki/Verlorenes_Update.

Verlorenes Update (auch englisch **lost update**) bezeichnet in der Informatik einen Fehler, der bei mehreren **parallelen Schreibzugriffen** auf eine gemeinsam genutzte Information

auftreten kann. Wenn zwei Transaktionen dieselbe Information verändern, dann können die Änderungen der ersten sofort durch die Änderungen der zweiten überschrieben werden.

1. create Qt console app: **OS01-LostUpdate**
2. get this code running! see http://doc.qt.io/qt-5/qthread.html

```
#include <QThread>
#include <string>
#include <iostream>
using namespace std;

class LostUpdate : public QThread{
private:
  string s;

  // gemeinsamer Speicher: Hier eine static Variable
  static int count;

public:
  LostUpdate(string s){
    this->s=s;
  }

  void run() override {
    for (int i = 1; i <=5; i++) {
      // READ
      int temp;
      temp= count;
      cout << s << "READ: " << temp<<endl;
      //QThread::msleep(50);

      // INCR
      temp= temp + 1;
      cout << s << "INCR: " << temp<<endl;
      QThread::msleep(20);

      //WRITE
      count= temp;
      cout << s << "WRITTEN: " << temp<<endl;
      //QThread::msleep(20);
    }
  }
};
int LostUpdate::count=1; // static member


int main(int argc, char *argv[])
{
  LostUpdate* t1= new LostUpdate("THREAD1: ");
  LostUpdate* t2= new LostUpdate("\t\t\t\tTHREAD2: ");
```

```
  t1->start();
  t2->start();

  t1->wait();  //vgl. join
  t2->wait();

  return 0;
}
```

Listing 3: OS01-LostUpdate.cpp

1.1.3 Sperrsynchronisation, critical Section, Mutex, Semaphore

critical section see https://de.wikipedia.org/wiki/Kritischer_Abschnitt

In der Informatik ist ein kritischer Abschnitt (engl. ‚critical section') eine Menge von Anweisungen, in dem sich zu einer Zeit nur ein einziger Prozess/Thread aufhalten darf. Ähnlich einem Bahnübergang, der nur vom Schienenfahrzeug oder nur von Straßenfahrzeugen befahren werden darf, aber nicht von beiden Fahrzeugarten gleichzeitig.

The so-called 'Sperrsynchronisation' uses Mutex to support critical sections.

Mutex see https://de.wikipedia.org/wiki/Mutex and http://doc.qt.io/qt-5/qmutex.html

```
#include <QMutex>
...
QMutex mutex;
int number = 6;

void method1(){
  mutex.lock();
  number *= 5;
  number /= 4;
  mutex.unlock();
}

void method2(){
  mutex.lock();
  number *= 3;
  number /= 2;
  mutex.unlock();
}
```

Listing 4: QMutex example

Semaphore see https://de.wikipedia.org/wiki/Semaphor_(Informatik).

Semaphor(e) ist eine Datenstruktur zur Steuerung eines ausschließenden Zugriffs. Die allgemeine Bedeutung von Semaphor ist Signalmast (Formsignal bei der Eisenbahn).

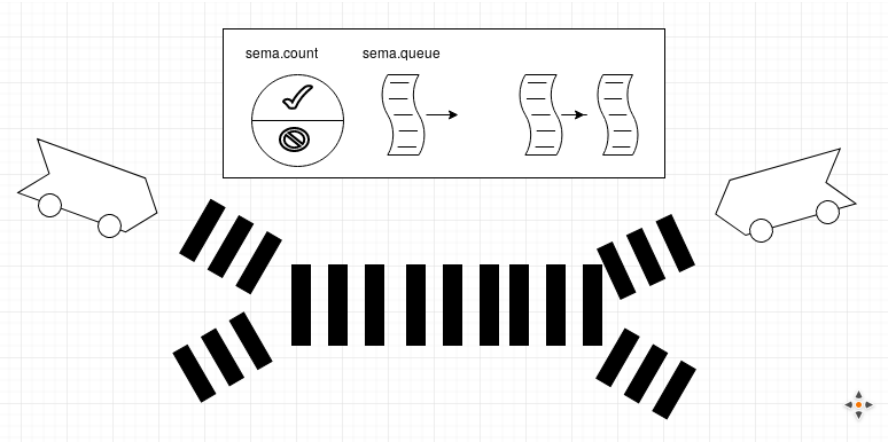

Figure 1: Semaphore Eisenbahnbeispiel

Wir betrachten in der Folge sog. binäre Semaphore, d.h. sie besitzen

1. einen Zähler, der die Werte 0 und 1 annhemen kann und
2. eine queue, um die Prozess-ID (PID) möglicher wartender Prozesse/Threads zu speichern.
3. P-Operation (auch wait() genannt)
4. V-Operation (auch notify() genannt)

```
1. DEFINE a SEMAPHORE
  typedef struct semaphor {
    int counter;
    Queue* queue; /* Warteschlange */
  } Semaphor;


2. INIT
void init(Semaphor* sema){
  sema->counter= 1;
  sema->queue= new Queue();
}


3. P-Operation/wait
   ----------------
void wait(Semaphor* sema){
  if (sema->counter > 0)
    sema->counter--;
  else
    // selbst PID in die Warteschlange einreihen und warten
    sema->queue->enqueue(PID);
}


4. V-Operation/notify
   ------------------
```

```
void notify(Semaphor* sema){
  if (sema->queue->isEmpty()){
    // Entblockieren eines Prozesses aus der Warteschlange
    PID= sema->queue->dequeue();
    resume(PID);
  }
  else
    sema->counter++;
}
```

Listing 5: semaphore example

1.1.4 Lab-QMutex: Lost-Update synchronized

see https://de.wikipedia.org/wiki/Mutex.

Qt::QMutex is the basic class for enforcing mutual exclusion. A thread locks a mutex in order to gain access to a shared resource. If a second thread tries to lock the mutex while it is already locked, the second thread will be put to sleep until the first thread completes its task and unlocks the mutex.

1. create Qt console app: **OS02-LostUpdate-Synchronized**
2. get this code running! see http://doc.qt.io/qt-5/qthread.html

```
/*
QMutex is the basic class for enforcing mutual exclusion. A thread locks a
    mutex in order to gain access to a shared resource. If a second thread
    tries to lock the mutex while it is already locked, the second thread
   will be put to sleep until the first thread completes its task and
   unlocks the mutex.
*/
#include <QThread>
#include <QMutex> // !!!!!!!!!!!!
#include <string>
#include <iostream>
using namespace std;

class LostUpdate : public QThread{
private:
  string s;

  // gemeinsamer Speicher: Hier eine static Variable
  static int count;

  // gegenseitiger Ausschluß !!!!!!!!!!!!!!!!!!!!!!
  // ***** ENTER CODE HERE ***** //
  static ________________________ mutex;

public:
  LostUpdate(string s){
```

```
    this->s=s;
  }

  void run() override {
    for (int i = 1; i <=5; i++) {

      // ***** ENTER CODE HERE ***** //

      // READ
      int temp;
      temp= count;
      cout << s << "READ: " << temp<<endl;
      QThread::msleep(50);

      // INCR
      temp= temp + 1;
      cout << s << "INCR: " << temp<<endl;
      QThread::msleep(20);

      //WRITE
      count= temp;
      cout << s << "WRITTEN: " << temp<<endl;
      QThread::msleep(20);

      // ***** ENTER CODE HERE ***** //

    }
  }
};

// static member
int LostUpdate::count=1;

// ***** ENTER CODE HERE ***** //
QMutex ____________________________;



int main(int argc, char *argv[]){
  LostUpdate* t1= new LostUpdate("THREAD1: ");
  LostUpdate* t2= new LostUpdate("\t\t\t\tTHREAD2: ");

  t1->start();
  t2->start();

  t1->wait();  //vgl. join
  t2->wait();

  return 0;
}
```

Listing 6: OS02-LostUpdate-Synchronized.cpp

1.1.5 Lab: Producer/Consumer FIFO

see https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

Producer/Consumer In computing, the producer–consumer problem is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

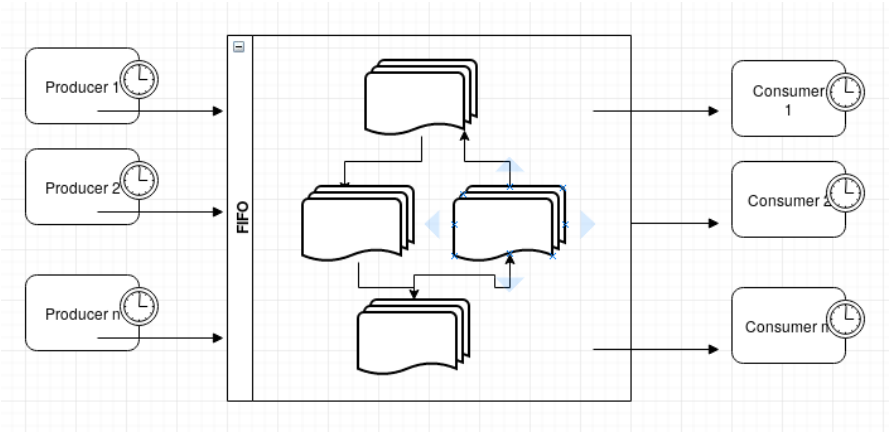

Figure 2: Producer/Consumer

FIFO First Step is to create class FIFO that store integer values and throws overfow/underflow exceptions.

Figure 3: FIFO-UML Klassendiagramm

1. create Qt console app: **OS03-FIFO**
2. get this code running! see http://doc.qt.io/qt-5/qthread.html

```
#include <QThread>
#include <stdexcept>
#include <iostream>

using namespace  std;

// ----------------------------------------
class FIFO {
private:
  int* queue;
  int size, begin, end;

public:
  FIFO(int size){
    this->size = size+1;
    this->queue = new int[size+1];  // !!!!!!!
    begin = 0;
    end = 0;
  }
  ~FIFO() { delete[] queue;}

  int dequeue()  {
    if (isEmpty())
      throw underflow_error("dequeue: underflow_error");

    // ***** ENTER CODE HERE *****

    return value;
  }

  bool isEmpty(){
    // ***** ENTER CODE HERE *****
  }

  bool isFull(){
    // ***** ENTER CODE HERE *****
  }

  void enqueue(int item)  {
    if(this->isFull()){
      throw overflow_error("enqueue: overflow_error");
    }
    else{
    // ***** ENTER CODE HERE *****
    }
  }
};
```

```
// ------------------------------------
class Producer : public QThread{
private:
  FIFO* fifo;
public:
  Producer(FIFO* fifo){
    this->fifo=fifo;
  }

 void produce(){
    for (int i=1; i<=5; i++){
      cout << "PRODUCER: enqueue " << i <<endl;
      fifo->enqueue(i);
    }
  }

 void run() override {
   produce();
 }
};


// ---------------------------------------
class Consumer : public QThread {
private:
  FIFO* fifo;

public:
  Consumer(FIFO* fifo){
    this->fifo=fifo;
  }

  void consume() {
    for (int i=1; i <=5; i++)
      cout << "\t\t\t\tCONSUMER: dequeue " << fifo->dequeue()<<endl;
  }
  void run() override {
    consume();
  }
};


int main(int argc, char *argv[]){

  FIFO* fifo= new FIFO(5);

  Producer* producer= new Producer(fifo);
  Consumer* consumer= new Consumer(fifo);
```

```
  producer->start();
  consumer->start();

  producer->wait();
  consumer->wait();

  return 0;
}
```

Listing 7: OS03-FIFO.cpp

Here are some figures to see how class FIFO works.

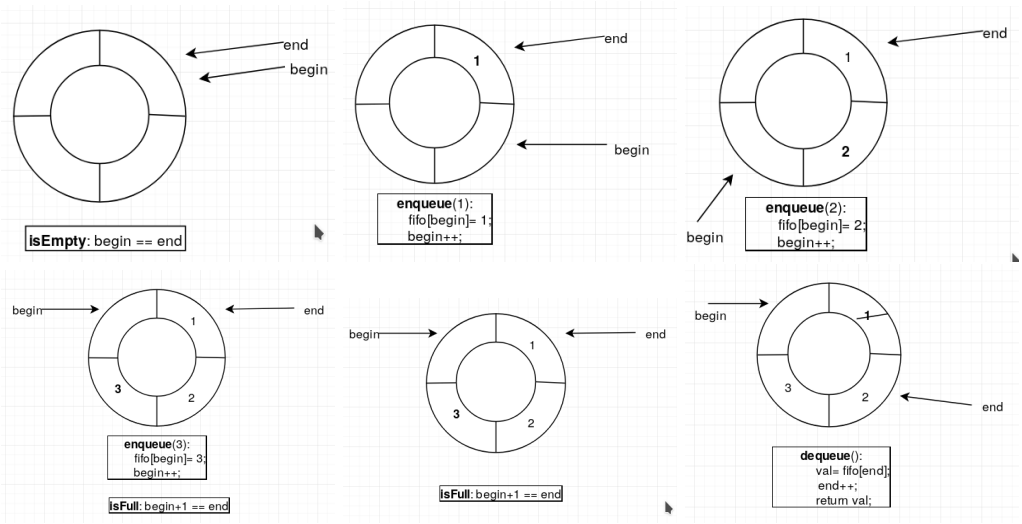

Figure 4: FIFO explained

1.1.6 Lab-QWaitcondition: Producer/Consumer FIFO synchronized

Synchronized FIFOs don't use exceptions if the buffer is full or empty, because they just have to wait for another producer/consumer process/thread to take/put an item into the FIFO. This kind of waiting can be implemented using so called condition variables.

Condition variables A condition variable manages a list of threads waiting until another thread notify them. Each thread that wants to wait on the condition variable has to acquire a lock first. The lock is then released when the thread starts to wait on the condition and the lock is acquired again when the thread is awakened.

A very good example is a concurrent FIFO (cylic Buffer). It's a cyclic buffer with a certain capacity with a begin and an end. Here finish the implementation of a Bounded Buffer using condition variables:

1. create Qt console app: **OS04-FIFO-Synchronized**
2. get this code running! see http://doc.qt.io/qt-5/qthread.html

```
#include <QThread>
#include <stdexcept>
```

```
#include <iostream>
#include <QMutex>
#include <QWaitCondition>
using namespace  std;

// ----------------------------------------
class FIFO {
private:
  int* queue;
  int size, begin, end;

  // Sperrsync
  static QMutex mutex;

  // Ereignissync:
  static QWaitCondition wc_isFull;
  static QWaitCondition wc_isEmpty;

public:
  FIFO(int size){
    this->size = size+1;
    this->queue = new int[size+1];  // !!!!!!!
    begin = 0;
    end = 0;
  }
  ~FIFO() { delete[] queue;}

  int dequeue()  {
    //    if (isEmpty())
    //      throw underflow_error("dequeue: underflow_error");

    mutex.lock(); // !!!!!!!!!!!!!!
    while(isEmpty())
      wc_isEmpty.wait(&mutex); // !!!!!!!!!!!!!!!

    int value = queue[end];
    end++;
    end %= size;

    wc_isFull.wakeAll();
    mutex.unlock(); // !!!!!!!!!!!!

    return value;
  }

  bool isEmpty(){
    if(begin == end){
      return true;
    }
    return false;
```

```
  }

  bool isFull(){
    if(end == (begin+1)%size){
      return true;
    }
    return false;
  }

  void enqueue(int item)  {

    // ***** ENTER CODE HERE *****

  }
};
// Sperrsync
QMutex FIFO::mutex;

// Ereignissync:
QWaitCondition FIFO::wc_isFull;
QWaitCondition FIFO::wc_isEmpty;


// -------------------------------------
class Producer : public QThread{
private:
  FIFO* fifo;
public:
  Producer(FIFO* fifo){
    this->fifo=fifo;
  }

  void produce(){
    for (int i=1; i<=5000; i++){
      cout << "PRODUCER: enqueue " << i <<endl;
      fifo->enqueue(i);
    }
  }

  void run() override {
    produce();
  }
};


// -------------------------------------
class Consumer : public QThread {
private:
  FIFO* fifo;
```

```
public:
  Consumer(FIFO* fifo){
    this->fifo=fifo;
  }

  void consume() {
    for (int i=1; i <=5000; i++)
      cout << "\t\t\t\tCONSUMER: dequeue " << fifo->dequeue()<<endl;
  }
  void run() override {
    consume();
  }
};


int main(int argc, char *argv[]){

  FIFO* fifo= new FIFO(5);

  Producer* producer= new Producer(fifo);
  Consumer* consumer= new Consumer(fifo);

  producer->start();
  consumer->start();

  producer->wait();
  consumer->wait();

  return 0;
}
```

Listing 8: OS04-FIFO-Synchronized.cpp

1.2 Exercises: Banking System, Eratosthenes

1.2.1 RDP-Aufgabe: Banking System - Lost Update

1. create Qt console app: **OS05-BANK-Synchronized**
2. get this code running! see http://doc.qt.io/qt-5/qthread.html

Die **Klasse SimpleBank** (hier java) besitzt das Array *konten*, das die Stände der einzelnen Konten enthält. Der Arrayindex soll als Kontonummer dienen. Für Transaktionen zwischen zwei Konten stellt die Bank die Methode *ueberweisung()* zur Verfügung. Ihr werden die beiden beteiligten Kontonummern sowie der Betrag der Überweisung übergeben.

```
class SimpleBank {
  // gemeinsamer Speciher
  private static int[] konten = {30, 50, 100};

  public void ueberweisung(int von, int nach, int betrag) {
    int neuerBetrag;
```

```
  // --> 1. ABHEBEN vom alten Konto
    neuerBetrag = konten[von];
    // Inkonsistenz, da neuer Betrag noch nicht vermerkt
    try {Thread.sleep((int)Math.random()*3000);}
    catch(InterruptedException e) {}

    neuerBetrag -= betrag;
    konten[von] = neuerBetrag;


  // --> 2. EINLEGEN aufs neue Konto
    neuerBetrag = konten[nach];
    // Inkonsistenz, da neuer Betrag noch nicht vermerkt
    try {Thread.sleep((int)Math.random()*3000);}
    catch(InterruptedException e) {}

    neuerBetrag += betrag;
    konten[nach] = neuerBetrag;
  }

  public void kontostand() {
    for(int i = 0; i < konten.length; i++)
      System.out.println("Konto "+ i +": " + konten[i]);
    }
  }
}
```

Listing 9: Bank (java)

Angestellte einer Bank machen Überweisungen. Damit mehrere Angestellte gleichzeitig Überweisungen vornehmen können, werden sie von Thread abgeleitet. Jeder Angestellte gehört zu einer Bank. Deshalb wird dem Konstruktor ein Verweis auf Bank übergeben. Über diesen Verweis wird die Methode ueberweisung() aufgerufen. Die beiden beteiligten Kontonummern sowie der Betrag werden dem Konstruktor ebenfalls übergeben und in entsprechenden privaten Membern der Klasse Angestellte gespeichert.

```
class Angestellter extends Thread {

  private SimpleBank bank;
  private int von, nach, betrag;

  public Angestellter(SimpleBank bank,  int von,
                      int nach, int betrag) {
    this.bank = bank;
    this.von = von;
    this.nach = nach;
    this.betrag = betrag;
  }

  public void run() {
```

```
  // Überweisung vornehmen
    bank.ueberweisung(von, nach, betrag);

  // Kontostand ausgeben
    System.out.println("Nachher:");
    bank.kontostand();
  }
}
```

Listing 10: Angestellter (java)

Das **Demonstrationsprogramm** vereinbart drei Verweise auf die Klasse Angestellter. Anschließend wird ein Bank-Objekt erzeugt und eine Übersicht über den Anfangsstand der Konten gegeben. Dann werden die drei Thread-Objekte erzeugt. Die Konten werden hierbei so gewählt, dass sich eine ringförmige Überweisung ergibt. Wenn alles ordnungsgemäß verläuft, dann müsste also die Kontenübersicht am Ende genauso aussehen wie am Anfang.

```
public class SimpleBankDemo {

  public static void main(String[] args) {
    Angestellter A1, A2, A3;
    SimpleBank b = new SimpleBank();

    System.out.println("Vorher: 20 Euro überweisen");
    System.out.println("        Konto_0->Konto_1->Konto_2->Konto_0");
    b.kontostand();

    // Eine ringförmige Überweisung
    A1 = new Angestellter(b, 0, 1, 20);
    A2 = new Angestellter(b, 1, 2, 20);
    A3 = new Angestellter(b, 2, 0, 20);

    A1.start();
    A2.start();
    A3.start();
  }
}
```

Listing 11: Demo Bank (java)

Wenn Sie das Programm starten, kann es sein, dass ein bestimmter Betrag 'verloren' geht. Man spricht vom **'LOST-UPDATE'-Problem**.

Aufgaben:

1. Bringen Sie das obige Programm zum Laufen.
2. Korrigieren Sie das obige Programm, sodass das LOST-UPDATE-Problem nicht mehr auftaucht.
3. Beschreiben Sie mit eigenen Worten das LOST-UPDATE-Problem und gehen Sie auf die Begriffe critical section, mutual exclusion ein.
4. Welche Art von Synchronisation wird hier angesprochen?

5. Beschreiben Sie in diesem Zusammenhang das Konzept der Semaphore-Variablen.

1.2.2 RDP-Aufgabe: Sieb des Eratosthenes - Producer/Consumer

Es soll ein Mechanismus der Primzahlenbestimmung nach dem sog. "Sieb des Eratosthenes" implementiert werden.

Grundprinzip dabei ist, dass jede ungerade Zahl der Reihe nach daraufhin untersucht wird, ob sie Vielfaches einer bereits erkannten Primzahl ist. Wenn das nicht der Fall ist, dann ist sie natürlich prim.

Die folgende Lösung stellt **für jede Primzahl einen eigenen Thread** zur Verfügung. D.h. wir erhalten eine Reihe von Threads, die als **Member je eine Primzahl** enthalten.

Der **Haupt-Thread erzeugt die ungeraden Zahlen** und sendet sie dem nächsten Thread in der Reihe.

Dieser ist für die **Primzahl 3** zuständig und filtert demnach alle Zahlen aus, die **Vielfache von 3** sind. D.h. er schickt diese Zahlen **nicht weiter**.

Alle anderen Zahlen reicht er an den nächsten Thread weiter, der für die Primzahl 5 zuständig ist, usw.

Jeder Thread erzeugt für die erste Zahl, die er weiterreichen muss (das ist die **nächste Primzahl**!), dynamisch seinen **Nachfolger-Thread**.

Aufgabe:

1. create Qt console app: **OS06-SIEB-ERATOSTHENES-producer-consumer**
2. get this code running! see http://doc.qt.io/qt-5/qthread.html

```
/** Worker der "Pipe" zur Primzahlenbestimmung
 * @author Roland Rössler
 */

class Worker extends Thread {
  /** die Primzahl, für die der Worker zuständig ist */
  private int myPrime;

  /** der nächste Worker in der "Pipe" */
  private Worker next=null;

  /** der Sende-/Empfangs-Puffer. Hier ein einf. Int Wert */
  private int buffer=0;

/** Erzeugung eines Workers und Zuordnung einer Primzahl
 * @param prime die Primzahl, für die der Worker zuständig ist
 */
  Worker(int prime) {
    myPrime=prime;
    next=null; /** nullsetzen*/
    System.out.println(this+" Prime: "+myPrime);
```

```
  }

/** "Arbeitsroutine" des Workers.
 * Es werden ungerade Zahlen empfangen, wenn diese
 * Vielfache der "eigenen" Primzahl sind, dann werden
 * sie "verschluckt", andernfalls an den nächsten
 * Worker weitergereicht
 */
  public void run() {
    int i = this.??????();     // Zahl empfangen
    while (i>0) {       //arbeite, solange i positiv ist
      if (i%myPrime != 0) {     // wenn i nicht Vielfaches von myPrime
        // erzeuge neuen Nachfolger-Thread oder
        // wenn er bereits existiert sende ihm
        // die Variable i

        if (next==null)
          ??????
        else
          ??????
      }

      i = ??????;                 // nächste Zahl holen
    }

    if (next!=null) ??????;   // Abbruchkennung weiterreichen
}

/** Senden einer Zahl an den Worker. <p>"send" wird von einem
 * anderem Thread aufgerufen.
 * @param i zu sendende Zahl
 */
  ?????? send(int i) {
    try {
      ?????? (buffer>0) ??????;        // warten bis Puffer frei

      buffer = i;                     // Zahl in Puffer eintragen

      ??????;                       // Empfänger benachrichtigen
    }
    catch (InterruptedException e) {}
  }

/** Empfangen einer Zahl.
 * "receive" wird vom aktuellen Worker aufgerufen.
 * @return empfangene Zahl
 */
  ?????? receive() {
    int result=0;
    try {
```

```
      ?????? ((result=buffer)==0)      // warten bis Zahl angekommen
        ??????;

      buffer=0;                        // Puffer freigeben

      ??????;                         // Sender benachrichtigen
    }
    catch (InterruptedException e) {}
    return result;
  }
}// end Worker


public class OS06_Eratosthenes{
  /** MAIN:Zahlengenerator.
   * Startet Worker für die Primzahl 3 u.
   * übergibt ihm alle ungeraden Zahlen ab 5
   */
  public static void main (String args[]) {
    Worker first = ??????;     // ersten Worker für Primzahl 3
    ??????;                    // starten

    int i=5;                         // nächste ungerade Zahl
    while (i<1000) {
      ??????;                  // ungerade Zahl an Worker senden
      i+=2;
    }

    ??????(-1);                   // Abschlusskennung senden
    System.out.println("finished");
  }
}
```

Listing 12: Eratosthenes (java)

Listings