PPM-java

Technical documentation

PH, 2017-01-13

1 Introduction

According to an Article on Wikipedia[1], "a peak programme meter (PPM) is an instrument used in professional audio for indicating the level of an audio signal." It is different from a normal VU meter in that it has a very short rise time (integration time) and a long return time. This allows audio producers to continouusly monitor the peaks of a programme signal. `ppm_java` implements a PPM type II which has a rise time of 23 dB in 10 ms and a fall time of -24dB in 2800ms.

The audio level maps to the meter scale as follows:

Input level [dB]		Meter mark	
min	**max**	**min**	**max**
-130	-24	0	1
-24	-20	1	2
-20	-16	2	3
-16	-12	3	4
-12	-8	4	5
-8	-4	5	6
-4	0	6	7
0	...	7	7

1 https://en.wikipedia.org/wiki/Peak_programme_meter

2 Running

System requirements

`ppm-java` requires

- Java (Tested with Java 1.8.)
- JackD audio server (Tested with version 1.9.10)

Test setup was a Linux machine (Kubuntu 14); program hasn't been tested on other platforms.

Example session

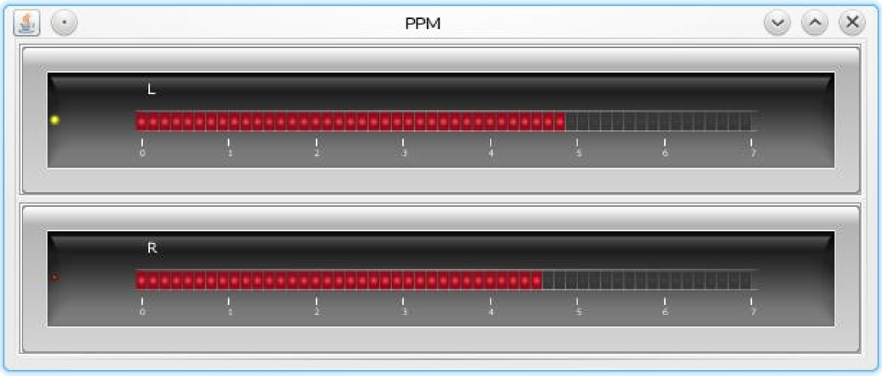
If JackD isn't running, start it:

```
user@local:~$ /usr/bin/jackd -r -dalsa -dhw:0 -r44100 -p1024 -n3 -Xraw
```

Start **`ppm-java`** from the commandline. We'll run it with the graphical linear gauge frontend:

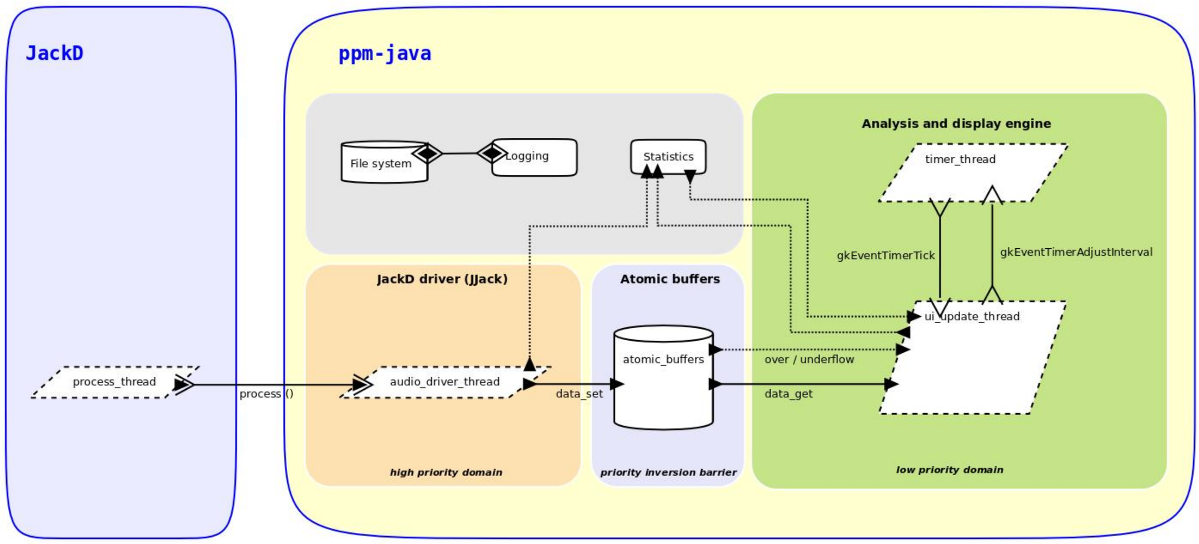
```
user@local:~$ java -jar /path/to/ppm.jar -u guiLinear -l /var/log/ppm.log
```

This will bring up a horizontal linear gauge. If the meter is receiving audio data the gauge will show the current audio levels:



3 Implementation

Design overview



ppm-java continouusly receives chunks of audio samples from the Jack server, and, for each chunk,

- computes the absolute peak value,
- converts it from raw sample values to decibels,
- applies a stepping integrator (PPM ballistics),
- and displays the result on a frontend.

Two activites are happening here: On the one hand, Jack delivers sample chunks, and, on the other hand, the program updates a display. These are two distinct activities, and both run in separate threads. The Jack side of things is handled by the audio driver thread, and everything else runs within the analysis and display engine thread.

The audio driver thread is time critical as it integrates with the Jack server. This means we cannot afford to have the audio driver run slow on us, as this would affect the Jack server and potentially the whole ecosystem of clients connected with the Jack server. The analysis and display engine (GUI side) not time critical - if it hangs for a few moments it's user annoying but at least it doesn't pull the entire Jack system down with it! For the program to run well we need to prevent priority inversion where the low priority thread (display engine) blocks the high priority thread (audio driver). We need to decouple the display engine from the audio driver in such a way that the audio driver won't get stuck (even if the display engine is frozen) and the display engine receives the audio data with minimal data loss.

Because of the need for decoupling we cannot send data directly from the audio driver to the display engine. Instead we use a bank of special buffers (one per channel) which receives data from the audio driver and holds it ready for the display engine when it fetches the data. The buffers are designed such that setting and fetching are atomic, i.e. only one thread a time can read/write data to the buffer (That's why we call them 'atomic buffers'). The bank of buffers acts as a priority inversion barrier between the display engine and the audio driver.

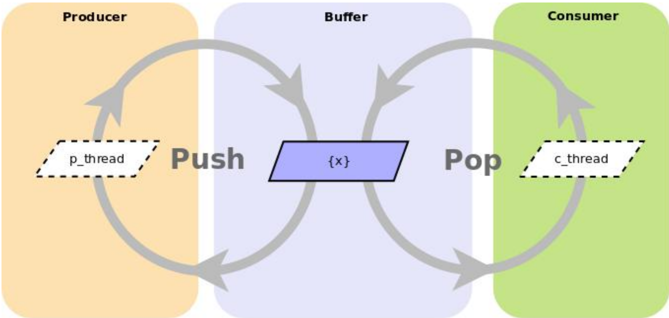
Challenges

- Good balance between scalability and fitness for the purpose. The challenge is to create a framework which is *well suited for the task* whilst being *generic enough to accomodate further development*.[2]
- Making two parallel activities with different priorities work together. On one side we have got a high priority activity (the **`jackd`** process), on the other side we've got a low priority activity (user interface updates). Both sides run independently of each other, i.e. in separate threads. *We have to avoid priority inversion* where the low priority thread (user interface updates) blocks the high priority thread (**`jackd`**).
- Minimizing data loss. The high priority **`jackd`** thread needs to work need for a protection against priority inversion brings up the possibility of data loss. For example we get data loss if the user interface thread blocks, whilst the **`JackD`** thread keeps pushing audio data. The challenge is to build in some *mechanism that minimizes data loss*.
- Keeping performance up. We need a framework which is *able to process the audio data without much latency*, so that updates on the meter happen almost immediately after the corresponding audio has come into the meter. There's always some latency with any audio application, but for a a metering solution the latency should be less than 25ms.

2 This is a hard balancing act - it's tempting to totally overengineer and create micromodules or to go the shortcut route and hack some contorted monolithic construction. In both cases, half a year later even the author won't have a clue what this thing does!

4 Multithreading

ppm-java makes extensive use of multithreading. The atomic buffer (class **TAtomicBuffer**) is an example of this. This class stores the audio samples coming from the input side and releases them to the GUI side when requested. Setting data and fetching data is done by two differnt threads, at the same time. One thread constantly pushes data to the buffer, whilst the other thread constantly fetches it from the buffer. This is a classic producer/consumer scenario.



For **ppm-java** we use multithreading in a slightly alternative way in how we make multiple threads work together without forcing wait time on them.

Without precautions, if multiple threads access the same object we can get surprising results. The following program has two theads simultaneously access a shared buffer. One thread pushes data, the other thread pops it[3]. No precautions are taken to make this program "multithread-proof".

Main program:

```
package ppm_java._dev.concept.example.multithread.unsafe;
public class TDev_Example_multithread_unsafe
{
    public static void main (String[] args)
    {
        TDev_Example_multithread_unsafe             unsafeClient;

        unsafeClient = new TDev_Example_multithread_unsafe ();
        unsafeClient.start ();
    }

    private TThreadConsumer             fConsumer;
    private TThreadProducer             fProducer;
    private int                         fValue;

    /**
     *
     */
    public TDev_Example_multithread_unsafe ()
    {
        fConsumer   = new TThreadConsumer (this);
        fProducer   = new TThreadProducer (this);
        fValue      = 0;
    }

    public void start ()
    {
        fConsumer.start ();
        fProducer.start ();
    }

    public void Pop ()
    {
        System.out.println ("Entering Pop  ().  Number: " + fValue);
        if (fValue > 0)
        {
            fValue--;
            try {Thread.sleep (500);} catch (InterruptedException e) {}
        }
        System.out.println ("Exiting  Pop  ().  Number: " + fValue);
```

3 Note that, in this program, **push** and **pop** just increment/decrement a counter. The point of the demo still holds.

```
    }

    public void Push ()
    {
        System.out.println ("Entering Push ().  Number: " + fValue);
        if (fValue < 1)
        {
            fValue++;
            try {Thread.sleep (500);} catch (InterruptedException e) {}
        }
        System.out.println ("Exiting  Push ().  Number: " + fValue);
    }
}
```

Producer:

```
package ppm_java._dev.concept.example.multithread.unsafe;

class TThreadProducer extends Thread
{
    private TDev_Example_multithread_unsafe             fHost;

    public TThreadProducer (TDev_Example_multithread_unsafe host)
    {
        fHost = host;
    }

    @Override
    public void run ()
    {
        int  i;
        int  delay;

        delay = 0;
        for (i = 1; i <= 10; i++)
        {
            fHost.Push ();
            delay += 10;
            try {Thread.sleep (delay);} catch (InterruptedException e) {}
        }
    }
}
```

Consumer:

```
package ppm_java._dev.concept.example.multithread.unsafe;

class TThreadConsumer extends Thread
{
    private TDev_Example_multithread_unsafe             fHost;

    public TThreadConsumer (TDev_Example_multithread_unsafe host)
    {
        fHost = host;
    }

    @Override
    public void run ()
    {
        int  i;
        int  delay;

        delay = 200;
        for (i = 1; i <= 10; i++)
        {
            fHost.Pop ();
            delay -= 10;
            try {Thread.sleep (delay);} catch (InterruptedException e) {}
        }
    }
}
```

The output of the program shows that Push() and Pop() are executed at arbitrary times. The producer and consumer just tread on each other's toes. The result is messed up data.

```
Entering Push ().  Number: 0
Entering Pop  ().  Number: 0     <--- Pop() should not be executed here.
Exiting  Push ().  Number: 0     <--- Number should be 1.
Exiting  Pop  ().  Number: 0
Entering Push ().  Number: 0
```

```
Entering Pop  ().  Number: 1
Exiting  Push ().  Number: 0
Entering Push ().  Number: 0
```

We need to get the threads to cooperate.

Thread cooperation primitives.

Since version 1.0 Java has support for multithreaded applications. Most notably, the language itself provides the **`synchronized`** keyword. This keyword allows us to make methods or code blocks thread-safe. The **`synchronized`** keyword creates a protected section of code that can only be entered by one thread a time. Such a section is called a *critical* section.

To make our previous example multithread proof we need to declare the main program's **`Push()`** and **`Pop()`** methods as **`synchronized`**.

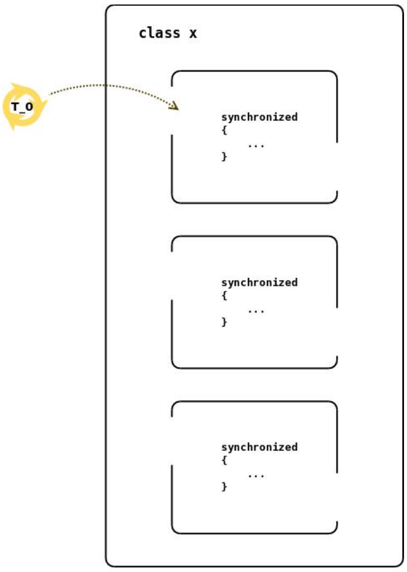
```
    public synchronized void Pop ()
    {
        System.out.println ("Entering Pop  ().  Number: " + fValue);
        if (fValue > 0)
        {
            fValue--;
            try {Thread.sleep (500);} catch (InterruptedException e) {}
        }
        System.out.println ("Exiting  Pop  ().  Number: " + fValue);
    }

    public synchronized void Push ()
    {
        System.out.println ("Entering Push ().  Number: " + fValue);
        if (fValue < 1)
        {
            fValue++;
            try {Thread.sleep (500);} catch (InterruptedException e) {}
        }
        System.out.println ("Exiting  Push ().  Number: " + fValue);
    }
```

This creates order, as both threads politely wait for each other! Now, our data is correct:

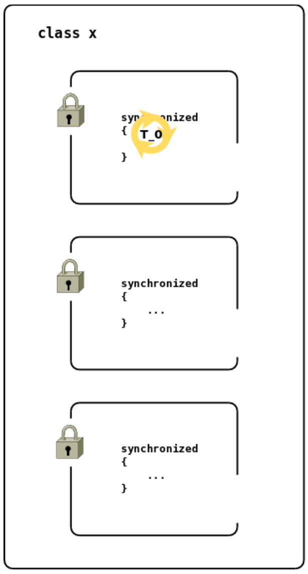
```
Entering Pop  ().  Number: 0
Exiting  Pop  ().  Number: 0
Entering Push ().  Number: 0
Exiting  Push ().  Number: 1
Entering Pop  ().  Number: 1
Exiting  Pop  ().  Number: 0
Entering Push ().  Number: 0
Exiting  Push ().  Number: 1
Entering Pop  ().  Number: 1
Exiting  Pop  ().  Number: 0
```

The **`synchronized`** keyword turns a method (or block of code) into a critical section. When a thread executes such a section then any other threads trying to execute the same section is made to wait until the first thread has finished. This is the reason why the synchronization in our example program works. If the producer thread has entered the **`Push()`** method, and the consumer tries to enter the **`Pop()`** method before the producer has finished, then the consumer must wait until the producer has left the **`Push()`** method. Both, **`Push()`** and **`Pop()`** are synchronized on the same object. The object is the lock; once it's acquired by a thread, no other thread can acquire it until the first thread has released the lock. This forces the producer and the consumer to wait for each other, ensuring data integrity. In a picture - the object is a room, and the threads are two storytellers in the room. Synchronization means only one of them can speak at a time.

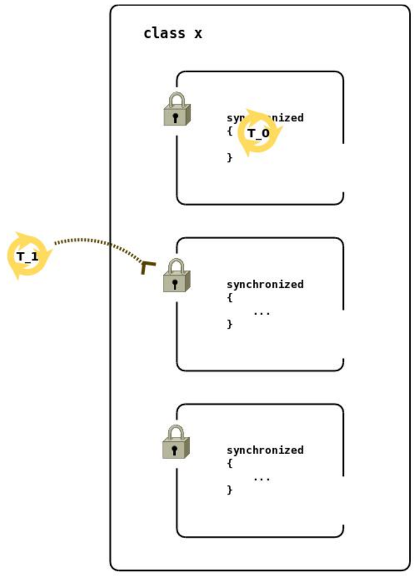
Note that locking works object wide: If one synchronized part in an object is locked, then all of the other synchronized parts are locked as well.



Thread T_0 will execute a synchronized section in class x.



Thread T_0 has entered the section. This will lock down all synchronized sections in class x.



Thread T_1 tries to enter another synchronized section in class x. However, thread T_0 is still executing it's section, so all other sections are still locked. Thread T_1 will be suspended until thread T_0 has finished executing it's section.

Wait problem

Synchronization means waiting. If one thread enters a synchronized section, all other threads have to wait (a thread trying will be suspended). This can block a thread for an uneccessarily long time. Here is an example which demonstrates the problem of the long wait:

Main program:

```
package ppm_java._dev.concept.example.multithread.wait;

public class THouse
{
    private static final int        gkNumVisitors   = 5;
    private static final long       gkTimeAudience  = 1000;

    public static void main (String[] args)
    {
        THouse              house;

        house = new THouse ();
        house.GetMeSomeVisitors ();
    }

    private TVisitor[]              fVisitors;

    public THouse ()
    {
        int i;

        fVisitors = new TVisitor [gkNumVisitors];
        for (i = 0; i < gkNumVisitors; i++)
        {
            fVisitors[i] = new TVisitor (this, i);
        }
    }

    public void GetMeSomeVisitors ()
    {
        int i;

        for (i = 0; i < gkNumVisitors; i++)
```

```
        {
            fVisitors[i].start ();
        }
    }

    public synchronized void Visit (int id)
    {
        try {Thread.sleep (gkTimeAudience);} catch (InterruptedException e) {}
    }
}
```

Visitor:

```
package ppm_java._dev.concept.example.multithread.wait;

class TVisitor extends Thread
{
    private THouse                  fHouse;
    private int                     fID;
    private String                  fPreamble;

    public TVisitor (THouse house, int id)
    {
        fHouse          = house;
        fID             = id;
        fPreamble       = "Visitor #" + fID;
    }

    public void run ()
    {
        long            t0;
        long            t1;
        long            dT;

        t0 = System.currentTimeMillis ();
        fHouse.Visit (fID);
        t1 = System.currentTimeMillis ();
        dT = t1 - t0;

        System.out.println (fPreamble + ": Had an audience! Time spent: " + dT + "ms.");
    }
}
```

A group of visitors would like to visit a house to have an audience with the home owner. Each audience with the owner is restricted to exactly one second. The owner speaks to one visitor a time. All others have to wait. Unfortunately, the output shows excessive waiting times for most visitors:

```
Visitor #0: Had an audience! Time spent: 1000ms.    <--- Thread #0 has a great deal: 0s wait time!
Visitor #4: Had an audience! Time spent: 2000ms.
Visitor #3: Had an audience! Time spent: 3000ms.
Visitor #2: Had an audience! Time spent: 4000ms.
Visitor #1: Had an audience! Time spent: 5001ms.    <--- Thread #1 has the worst deal: 4s wait time!
```

The problem is the synchronization mechanism: When one thread acquires a lock, all other threads trying to acquire the same lock will be suspended until the lock is released. And that can be a long time waiting in suspense. For a realtime thread such as the audio driver in **`ppm-java`**, any wait is prohibitive.

Wait free access

Since version 1.5 Java offers atomic variables in the package **`java.util.concurrent.atomic`**. Atomic variables are variables that can be manipulated with one single processor instruction. As a side effect, this makes them inheritly thread-safe[4].

`java.util.concurrent.atomic` offers several types of atomic variables, but of special interest is the type **`java.util.concurrent.atomic.AtomicInteger`**, and especially it's method **`compareAndSet()`**). The proper signature of this method is

4 With plain variables, thread safety is not guaranteed. For example, with a plain variable a simple assignment may need multiple processor instructions to complete which opens it up to data corruptions if manipulated by multiple threads simultaneously.

```
public final boolean compareAndSet(int expect, int update)
```

`compareAndSet` atomically sets the value to **`update`** if the current value equals **`expect`**. The method returns **`true`** if the value was successully changed and **`false`** if the value change failed. This method makes it useful to coordinate multiple threads without locking them up. Here is the visitor program again, this time using an **`AtomicInteger`** as coordination primitive:

Main program:

```
package ppm_java._dev.concept.example.multithread.waitfree;

import java.util.concurrent.atomic.AtomicInteger;

public class THouse
{
    private static final int        gkNumVisitors   = 5;
    private static final int        gkTimeAudience  = 1000;
    private static final int        gkLocked        = 1;
    private static final int        gkUnlocked      = 0;

    public static void main (String[] args)
    {
        THouse              house;

        house = new THouse ();
        house.GetMeSomeVisitors ();
    }

    private AtomicInteger           fState;
    private TVisitor[]              fVisitors;

    public THouse ()
    {
        int i;

        fState    = new AtomicInteger (gkUnlocked);
        fVisitors = new TVisitor [gkNumVisitors];
        for (i = 0; i < gkNumVisitors; i++)
        {
            fVisitors[i] = new TVisitor (this, i);
        }
    }

    public void GetMeSomeVisitors ()
    {
        int i;

        for (i = 0; i < gkNumVisitors; i++)
        {
            fVisitors[i].start ();
        }
    }

    public boolean Visit (int id)
    {
        boolean isSuccess;

        isSuccess = fState.compareAndSet (gkUnlocked, gkLocked);
        if (isSuccess)
        {
            try {Thread.sleep (gkTimeAudience);} catch (InterruptedException e) {}
            fState.getAndSet (gkUnlocked);        // atomic setting
        }

        return isSuccess;
    }
}
```

Visitor:

```
package ppm_java._dev.concept.example.multithread.waitfree;

class TVisitor extends Thread
{
    private THouse                  fHouse;
    private int                     fID;
    private String                  fPreamble;
```

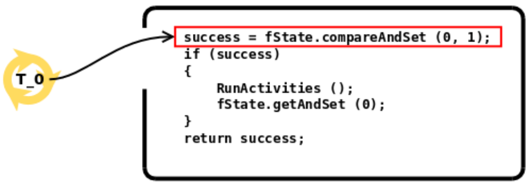
```
    public TVisitor (THouse house, int id)
    {
        fHouse          = house;
        fID             = id;
        fPreamble       = "Visitor: " + fID;
    }

    public void run ()
    {
        long            t0;
        long            t1;
        long            dT;
        long            tTot;
        boolean         isSuccess;

        tTot = 0;
        do
        {
            t0        = System.currentTimeMillis ();
            isSuccess = fHouse.Visit (fID);    // Costs next to no time if fHouse is busy, i.e. no suspending!
            t1        = System.currentTimeMillis ();
            dT        = t1 - t0;
            tTot     += dT;
            if (isSuccess)
            {
                System.out.println (fPreamble + ": Had an audience! Time spent (total): " + tTot + "ms.");
            }
            else
            {
                // Couldn't get an audience this time. At least, my thread is not
                // suspended. I'll go and do something else and try again later.
                try {Thread.sleep (500);} catch (InterruptedException e) {}
            }
        }
        while (! isSuccess);
    }
}
```

The output shows that there's a lot less waiting than when using the **`synchronized`** keyword. Now, each visitor spends zero seconds of wait time[5]:

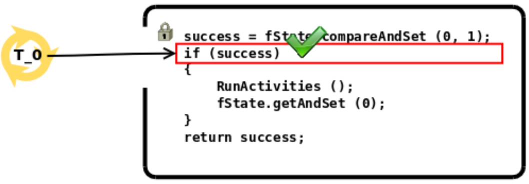
```
Visitor: 1: Had an audience! Time spent (total): 1000ms.
Visitor: 0: Had an audience! Time spent (total): 1000ms.
Visitor: 2: Had an audience! Time spent (total): 1000ms.
Visitor: 3: Had an audience! Time spent (total): 1001ms.
Visitor: 4: Had an audience! Time spent (total): 1000ms.
```

The advantage of this way of thread coordination is that no thread gets ever suspended, and all threads will continue to work, also those which could not enter a critical section. Whilst the synchronized primitive simply suspends any thread trying to enter a protected and occupied critical section. Besides, some sources say that synchronization is much more expensive than atomic variables as the JVM does a lot of work in the background to operate it. Therefore, atomic variables incur less of a performance penalty than synchronized parts.

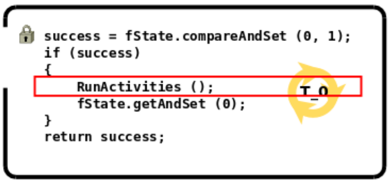
5 It may be confusing to you that it says “zero seconds of wait time” whilst the visitor thread does a **`Thread.sleep(500)`** when the **`isSuccess`** flag is **`false`**. Isn't **`Thread.sleep()`** a wait as well? In this context, it isn't! Because that sleep is not enforced from outside (via the **`synchronized`** mechanism), but chosen by the thread! Instead of sleeping the visitor could have done some other tasks, e.g. going for a walk, doing some shopping,... in the wait-free scenario it's up to the visitor thread what it does whilst waiting for the audience. With **`synchronized`** the sleep is enforced upon the visitor, i.e. the receptionist knocks him out.



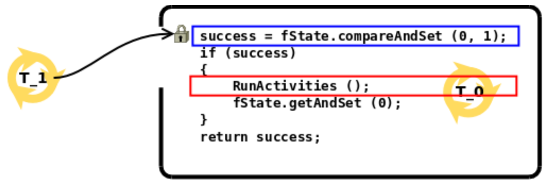
1. Thread T_0 queries the critical section lock. This invokes the atomic compareAndSet method. Only T_0 can access fState at that moment. Since the lock was OPEN (value 0), it was possible to set it to CLOSED (value 1).



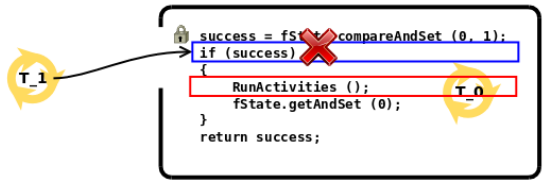
2. fState has been successfully set to CLOSED. Hence, success is TRUE and T_0 can enter the critical section.



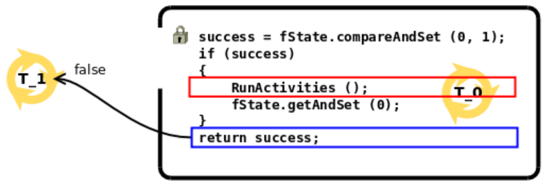
3. T_0 has entered the critical section and is doing it's work.



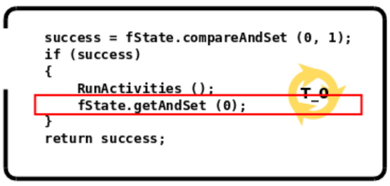
4. Meanwhile, thread T_1 queries the critical section lock.



5. Attempt failed to set fState to CLOSED. Therefore, success is FALSE and T_1 can't enter the critical section.



6. The return value tells T_1 that it couldn't enter the critical section. T_1 stays running (never suspended), can get on with other things and come back later to try again. For T_1 the whole ordeal took a few nanoseconds, and no time suspended (wait free).



7. T_0 has finished it's work and sets fState back to OPEN. getAndSet is atomic, i.e. no thread could use compareAndSet at precisely the same time.

5 Modules

Behind the simple purpose of this program is a modular machinery implementing it. Various modules work together, each one performing a small sub task. The various modules can be connected; once done they work together as a complete whole.

The modular design increases complexity initially, but makes the design much more flexible. For example, all frontends except the **`consoleText`** frontend include PPM ballistics, and due to the modular nature this special case was easy to accommodate. Also, a modular design aids development of new features. For example, the user might be interested in a history view to see the last minute of level measurements. Or, the meter could be used as a silence detector, sending an alarm event somewhere if the audio level falls below a threshold for a prolonged time.

At the moment, **`ppm-java`** comprises the following modules:

`TAudioContext_JackD`	The audio driver (backend).
`TNodePump`	Data pump. Fetches audio data from the attached atomic buffer and passes it on to the next module. Data is fetched and passed on each time a **`gkEventTimerTick`** is received.
`TNodePeakEstimator`	Calculates the absolute peak value of a sample chunk. Being absolute, the value is always positive.
`TNodeConverterDb`	Converts a sample (peak) value from raw level to dB. Conversion according to $y = 20 \log_{10} (x)$ where x: normalized absolute sample value [Vnorm], $0 <= x <= 1$ y: sample value [dB]
`TNodeIntegrator_PPMBallistics`	A stepping approximator to emulate PPM ballistics.
Derivatives of **`VFrontend`**	The frontend used for the current session.

Backend

ppm-java runs as a Jack[6] client. The Jack server connects audio programs and hardware so they can communicate with each other. Jack runs in it's own process, **jackd**. This means that there must be a running instance of **jackd** before starting **ppm-java**. Once both of them are running, **ppm-java** will find the running **jackd** server and register as a client. The user can then connect **ppm-java** to outputs of any other clients registered with **JackD** and display the audio levels on a frontend.

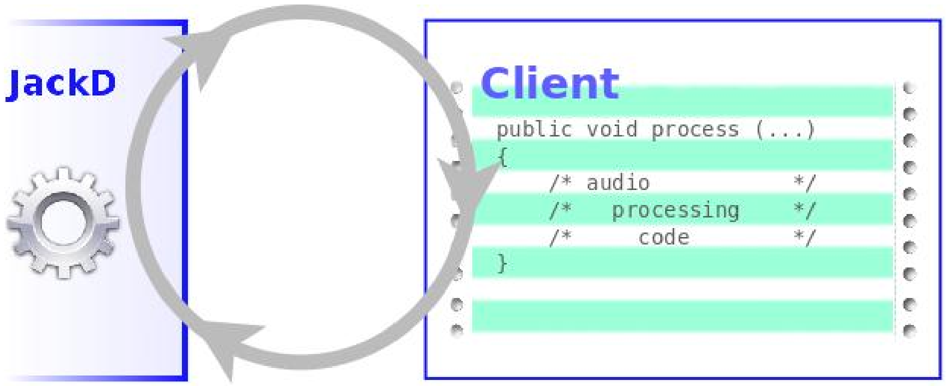
Through the Jack audio server, ppm-java can receive audio data from attached hardware or from other Jack compatible programs. All communication with Jack is encapsulated in the module **TAudioContext_JackD** which relies on the **JJack** library[7] for the actual connection with Jack.

JJack makes it easy to write Jack clients. Here's a client that receives audio from a port on the Jack server and prints peak sample values on the console (full program, see appendix 1, listing 1):

```
package ppm_java._dev.concept.trial.JJack;

import... /* Various imports */

/**
 * Loads the JJack driver, connects to a running instance of JackD and prints
 * the peak of every incoming frame to stdout.
 *
 * @author peter
 */
public class TDev_Trial_JJack_process_01 implements JJackAudioProcessor
{
    public static void main (String[] arge) {/* Setup stuff... */}

    private int fIFrame = 0;
    public TDev_Trial_JJack_process_01 ()
    {
        fIFrame = 0;
    }

    public void process (JJackAudioEvent e)
    {
        FloatBuffer         inBuf;
        int                 nSamples;
        int                 i;
        float               s;
        float               sRect;
        float               peak;


        fIFrame++;
        inBuf       = e.getInput (0);
        nSamples    = inBuf.limit ();
        peak        = 0;
        for (i = 0; i < nSamples; i++)
        {
            /* Get next sample. */
            s = inBuf.get (i);

            /* Rectify (i.e. mirror a negative sample to it's positive opposite). */
            sRect = (s < 0)  ?  -s : s;

            /* Determine peak value. */
            peak = (sRect > peak)  ?  sRect : peak;
        }
        System.out.println ("Frame: " + fIFrame + "; Peak value (abs): " + peak + ".");
    }
}
```

6 http://www.jackaudio.org
7 https://sourceforge.net/projects/jjack

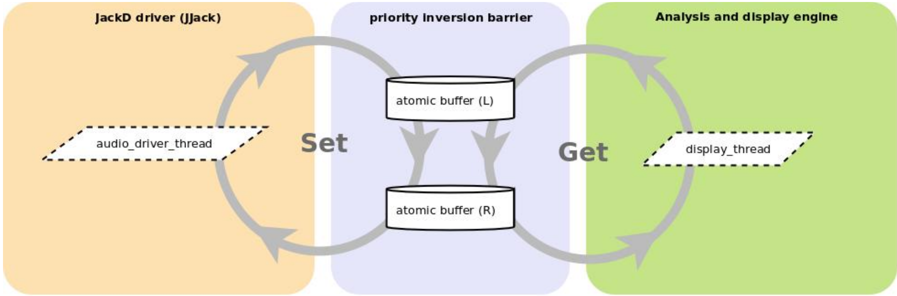
The central piece of this code is the method **`process(JJackAudioEvent e)`**. The method receives one parameter of type **`JJackAudioEvent`** (provided by **`JJack`**) which contains everything needed to extract and push audio data from/to the Jack server. All the client's sample processing is done in the **`process(...)`** method. The Jack server (via the **`JJack`** bridge) calls this method in an endless cycle (several times per second) until the program ends.



The audio driver thread is time critical, i.e. a call to **`process(...)`** method must be guaranteed to complete in a fixed time shorter than the cycle time of the **`jackd`** instance. Here's the program output:

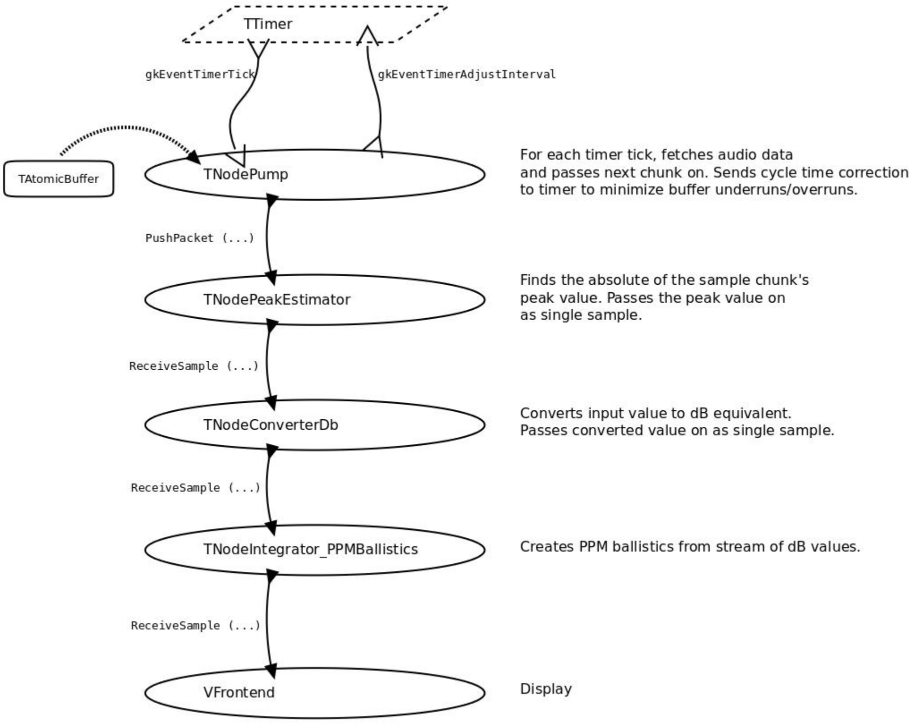
```
native jjack library loaded using system library path
natively registering jack client "JJack"
jack_client_new: deprecated
using 2 input ports, 2 output ports
Frame: 1; Peak value (abs): 0.0.
Frame: 2; Peak value (abs): 0.0.
Frame: 3; Peak value (abs): 0.0.
Frame: 4; Peak value (abs): 0.042320143.
Frame: 5; Peak value (abs): 0.036235876.
Frame: 6; Peak value (abs): 0.038238987.
Frame: 7; Peak value (abs): 0.038017016.
Frame: 8; Peak value (abs): 0.04076365.
Frame: 9; Peak value (abs): 0.022242684.
Frame: 10; Peak value (abs): 0.022647541.
```

In **`ppm-java`** we cannot push the data directly to the display engine. If the engine blocks whilst it receives data it would result in the audio driver being stuck in the **`process(...)`** method (priority inversion), with unknown side effects for the running **`jackd`** instance. To prevent this from happening we push the audio data to a bank of atomic buffers (one per audio channel) from where the display engine will fetch it. The atomic buffers are designed to stream data from a high priority thread to a low priority thread without ever blocking the high priority thread. This way we decouple the time critical audio driver thread from the low priority display engine thread. The display engine might still get stuck - but now the only consequence would be lost data. The audio driver (i.e. the jack server) will continue to work, even if the display engine freezes up.



Analysis and display engine

The main work of **`ppm-java`** is done inside the analysis and display engine. This device takes the raw audio data and converts it to a stream of decibel values to display them on a front end. The engine comprises multiple modules.



The analysis and display engine runs inside a low priority thread. It is connected to the audio driver backend by a bank of atomic buffers which ensure that the analysis and display engine won't slow down the audio driver thread. This comes at the price of data integrity. If the analysis and display engine does slow down (e.g. graphics takes longer than usual to do a refresh) then it will miss some of the data values coming from the audio driver. There isn't really any other way to resolve freeze-ups of the display engine than to simply drop the past data and carry on with the current data.

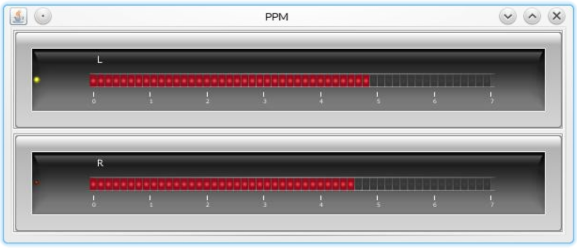
Frontends

Various frontends are available. Some offer graphical user interface, others work in text-only mode, i.e. in the console from which **`ppm-java`** has been started. Most of them include PPM ballistics in the display.

`guiRadial` A PPM lookalike.

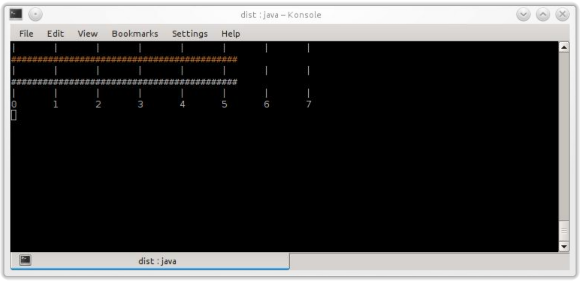
guiLinear

Horizontal linear bargraph.



consoleLinear

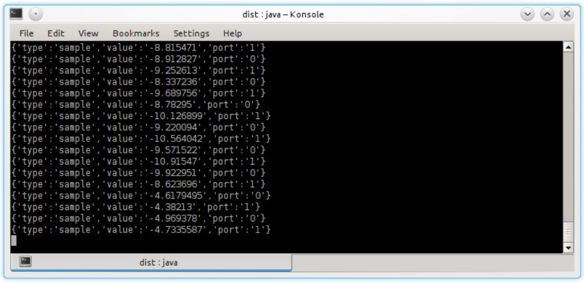
Horizontal linear bargraph on the console (text mode).



consoleText

Text stream to stdout. Data is presented in JSON format. This frontend is designed to stream audio level data to another application - e.g. via a pipe.

When **ppm-java** runs with this frontend, there will be no PPM ballistics, i.e. the peak values will be displayed directly, without integration.



6 Session setup

7 Events

8 Statistics interface

9 Class hierarchy

10 Roadmap

- Finish documentation (est. one week).
- True peak capabilities. So far, we just take the highest absolute sample value per GUI cycle and use that for the display. However, in reality most peaks fall between peaks on the actual waveform. This means our meter does not display the waveform peaks but only the sample peaks, resulting in an underread of typically 3dB. A truepeak module is needed to reduce the underread. We would design the truepeak module according to the recommended algorithm in BS.1770-4[8] (est one week).
- Port to C++/QT. The **`ppm-java`** program is a proof of concept. And it was simply easier to do the design in Java (e.g. we don't have to keep track of header files). Once the design has matured, it should be fairly straight forward to port the program to C++, using the QT toolkit and other support libraries (est. four weeks).

8 BS.1770-4, Annex 2. https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.1770-4-201510-I!!PDF-E.pdf

11 Appendices

App. 1: Example listings

Listing 1:

```
package ppm_java._dev.concept.trial.JJack;

import java.nio.FloatBuffer;
import de.gulden.framework.jjack.JJackAudioEvent;
import de.gulden.framework.jjack.JJackAudioProcessor;
import de.gulden.framework.jjack.JJackException;
import de.gulden.framework.jjack.JJackSystem;
import ppm_java.util.logging.TLogger;

/**
 * Loads the JJack driver, connects to a running instance of JackD and prints
 * the peak of every incoming frame to stdout.
 *
 * @author peter
 */
public class TDev_Trial_JJack_process_01 implements JJackAudioProcessor
{
    public static void main (String[] args)
    {
        TDev_Trial_JJack_process_01 processor;

        /* Setting up Jack client and terminator thread.  */
        processor  = new TDev_Trial_JJack_process_01 ();

        /* Connecting with the running instance of Jack */
        JJackSystem.setProcessor (processor);

        /* Terminator timer - must run in separate thread */
        new Thread ()
        {
            public void run ()
            {
                try
                {
                    Thread.sleep (500);
                    JJackSystem.shutdown ();
                }
                catch (JJackException | InterruptedException e)
                {
                    e.printStackTrace();
                }
                System.exit (1);
            }
        }.start ();
    }

    private int fIFrame;

    public TDev_Trial_JJack_process_01 ()
    {
        fIFrame = 0;
    }

    /**
     * The process callback. Called by the Jack server via the JJack bridge. Time critical!
     */
    @Override
    public void process (JJackAudioEvent e)
    {
        FloatBuffer         inBuf;
        int                 nSamples;
        int                 i;
        float               s;
        float               sRect;
        float               peak;

        fIFrame++;
        inBuf       = e.getInput (0);
        nSamples    = inBuf.limit ();
        peak        = 0;
        for (i = 0; i < nSamples; i++)
        {
            /* Get next sample. */
            s = inBuf.get (i);
```

```
            /* Rectify (i.e. mirror a negative sample to it's positive opposite). */
            sRect = (s < 0)  ?  -s : s;

            /* Determine peak value. */
            peak = (sRect > peak)  ?  sRect : peak;
        }
        System.out.println ("Frame: " + fIFrame + "; Peak value (abs): " + peak + ".");
    }
}
```

Glossary

Priority inversion

In computer science, priority inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks.

This violates the priority model that high priority tasks can only be prevented from running by higher priority tasks and briefly by low priority tasks which will quickly complete their use of a resource shared by the high and low priority tasks.

https://en.wikipedia.org/wiki/Priority_inversion

12 License

A program to display audio levels using PPM ballistics.