**Eindopdracht Practicum Operating Systems**
**Een multithreaded equalizer applicatie**

In deze afsluitende eindopdracht gaan we diverse multithreaded programmeertechnieken in de praktijk brengen. Het doel is om een multithreaded C++ applicatie te maken die toonregeling kan uitvoeren op de inhoud van een geluidsbestand.

De applicatie is een console applicatie **ate** (**A**ll **T**hings **E**qual) genaamd. De applicatie moet een aantal parameters van de command line herkennen volgens de volgende structuur:

```
ate –p:<number of threads>
    -b:<bass intensity>
    -t:<treble intensity>
    <input file> <output file>
```

De parameters hebben de volgende betekenis en mogelijke waarden:

| Parameter | Betekenis | Geldige waarden | Voorbeeld |
|---|---|---|---|
| -p:<n_o_t> | Het aantal threads dat de applicatie gebruikt. | integer<br>1 .. 8 | -t:4 |
| -b:<b_i> | De aanpassing van de intensiteit van de lage tonen in decibels (dB) | integer<br>-6 tot 6 | -b:-3 |
| -t:<t_i> | De aanpassing van de intensiteit van de hoge tonen in decibels (dB) | integer<br>-6 tot 6 | -t:+4 |
| <input file> | Het geluidsbestand waarop de toonregeling moet worden toegepast.<br><br>Dit moet een raw pcm-bestand zijn met de volgende kenmerken:<br>• Sample rate = 44100<br>• Bitdepth = 16 bits<br>• Aantal kanalen = 1 (ofwel mono) | *.pcm | C:\input.pcm |
| <output file> | Het uitvoerbestand met daarin de resultaten na equalizing. | *.pcm | C:\output.pcm |

Wanneer er te weinig of verkeerde parameters worden meegegeven moet de applicatie stoppen, een juiste foutmelding geven en bovenstaande *usage* afdrukken.

**TIP:** Je kunt de parameters uitlezen uit de argv[] string array die meegegeven wordt aan de main functie. In argc staat het aantal meegegeven parameters.

Nadat alle parameters zijn uitgelezen gaat de applicatie aan de slag om de toonregeling op het opgegeven geluidsbestand toe te passen. Om dit snel te laten verlopen worden meerdere threads gebruikt die elk een bepaalde stap op een deel (*block*) van het geluidsbestand uitvoeren.

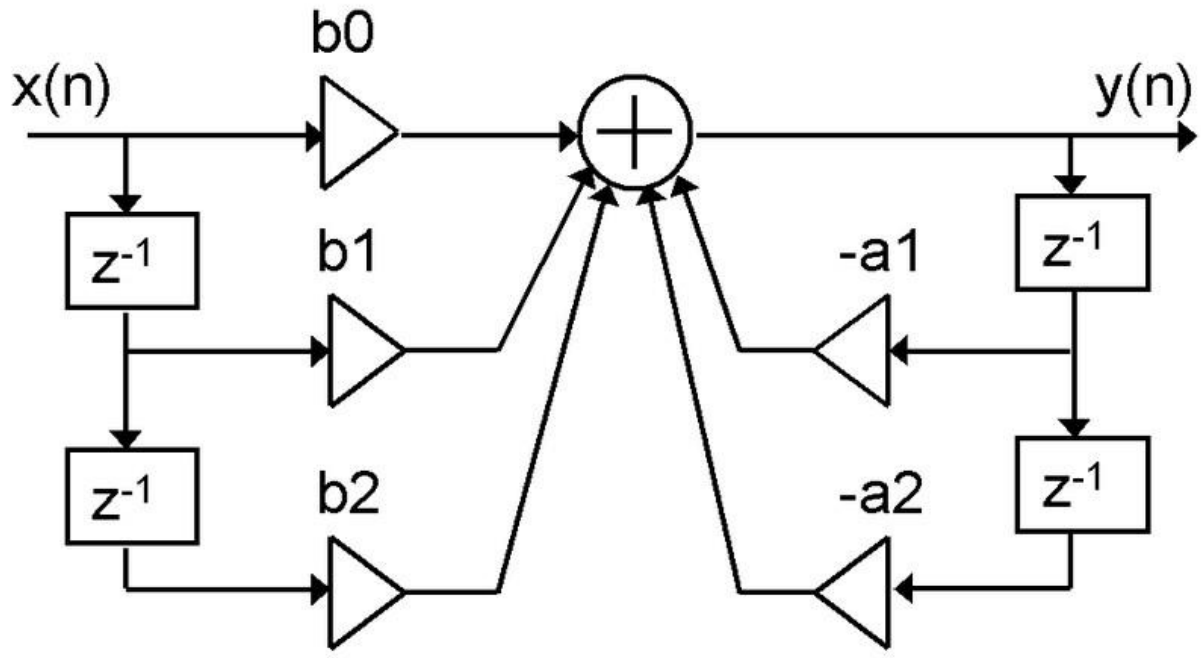
In het begin wordt het invoerbestand gelezen en vervolgens wordt de input in *blocks* verdeeld. Vervolgens worden deze *blocks* verwerkt. Elk *block* bevat 1024 samples en is voor een 16 bit mono bestand dus 2048 bytes groot. Op elk van de *blocks* moeten 3 losse bewerkingen worden uitgevoerd:

1. Toepassen van de bass equalizer
2. Toepassen van de treble equalizer
3. Wegschrijven van het resultaat naar het uitvoerbestand

De volgorde van de eerste 2 stappen is niet van belang. De derde stap moeten natuurlijk wel als laatste gezet worden. Je dient threads in te zetten om deze bewerkingen (optimaal) uit te voeren.

## De Biquad

Om de toonregeling te realiseren maken we gebruik van een Biquad. Dit is een veelgebruikte constructie in DSP-techniek (Digital Signal Procssing) om signalen te filteren. Hieronder zie je een schematische weergave van een Biquad:




x[n] is de array met input samples
y[n] is een nieuwe array met output samples

Iedere waarde van y[n] wordt berekent middels de volgende vergelijking:

y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2] + a1*y[n-1] + a2*y[n-2]

De waarde van een output sample is dus afhankelijk van de huidige input sample x[n] en de twee samples daarvoor x[n-1] en x[n-2]; alsmede van de vorige 2 output samples y[n1] en y[n-2].

Afhankelijk van de waarden van de coëfficiënten a1, a2, b0, b1 en b2 wordt een andere bewerking op het geluid gedaan.

Om te bepalen welke waarden je moet gebruiken voor de coëfficiënten b0, b1, b2, a1 en a2 kun je de volgende code gebruiken:

```
void bassCoefficients(int intensity, double *b0, double *b1, double *b2, double *a1, double *a2)
{
    double frequency = 330;
    double qFactor = 0.5;
    double gain = intensity;
    double sampleRate =44100;

   double pi=4.0*atan(1);
   double a=pow(10.0, gain/40);
   double w0=2*pi*frequency/sampleRate;
   double alpha=sin(w0)/(2.0*qFactor);
   double a0=(a+1)+(a-1)*cos(w0)+2.0*sqrt(a)*alpha;

   *a1=-(-2.0*((a-1)+(a+1)*cos(w0)))/a0;
   *a2=-((a+1)+(a-1)*cos(w0)-2.0*sqrt(a)*alpha)/a0;
   *b0=(a*((a+1)-(a-1)*cos(w0)+2.0*sqrt(a)*alpha))/a0;
   *b1=(2*a*((a-1)-(a+1)*cos(w0)))/a0;
   *b2=(a*((a+1)-(a-1)*cos(w0)-2.0*sqrt(a)*alpha))/a0;
}

void trebleCoefficients(int intensity, double *b0, double *b1, double *b2, double *a1, double *a2)
{
    double frequency =3300;
    double qFactor = 0.5;
    double gain = intensity;
    double sampleRate = 44100;
    double pi=4.0*atan(1);
    double a=pow(10.0, gain/40);
    double w0=2*pi*frequency/sampleRate;
    double alpha=sin(w0)/(2.0*qFactor);
    double a0=(a+1)-(a-1)*cos(w0)+2.0*sqrt(a)*alpha;

   *a1=-(2.0*((a-1)-(a+1)*cos(w0)))/a0;
   *a2=-((a+1)-(a-1)*cos(w0)-2.0*sqrt(a)*alpha)/a0;
   *b0=(a*((a+1)+(a-1)*cos(w0)+2.0*sqrt(a)*alpha))/a0;
   *b1=(-2.0*a*((a-1)+(a+1)*cos(w0)))/a0;
   *b2=(a*((a+1)+(a-1)*cos(w0)-2.0*sqrt(a)*alpha))/a0;
 }
```

De gebruikte coëfficiënten zijn voor alle samples dezelfde. Je hoeft ze dus maar 1 keer te laten berekenen aan het begin van je programma. Het programma gebruikt dan de berekende coëfficiënten om het geluid te bewerken.

In het programma moet de data in een *block* volgens de eerder genoemde vergelijking worden aangepast.

## De PCM-bestanden

Om je programma te testen kun je gebruik maken van het test-bestand op Brightspace of zelf een geschikt bestand maken met bijvoorbeeld Audacity. Zorg dat je een mono/16 bits/44100 geluidsbestand wegschrijft als uncompressed RAW (header-less) bestand met 16 bits signed als encoding.

Een PCM raw bestand heeft geen header. Het bestand is eenvoudigweg een opsomming van samples. Je moet dus van tevoren weten wat het formaat van de samples in het bestand is om ermee te kunnen werken. Voor deze opdracht zijn dit 16 bits integers.

**TIP:** Voor een 16 bits integer kun je in C++ het volgende type gebruiken:

```
signed short sample;
```

De maximale waardes die een signed short kan aannemen zijn:

–32,768 tot 32,767

## Stappen:

1. Maak een ontwerp op papier. Beantwoord daarin de volgende vragen:
   - Welke threads heeft je applicatie ?
   - Wat is de taak van elke thread ?
   - Welke informatie moet over elk *block* bekend zijn ?
   - Hoe wordt bijgehouden welke bewerkingen al op een *block* zijn uitgevoerd ?
   - Hoe ga je het aantal threads beperken tot het maximum dat is opgegeven ?
   - Hoe gaat een thread slapen indien er geen *block* meer beschikbaar is om te bewerken ?
   - Hoe zorg je dat de *blocks* in het uitvoerbestand in de juiste volgorde in het bestand terecht komen ?

2. Bespreek je ontwerp met de docent alvorens het te gaan implementeren
3. Implementeer de *Block* class en pas de *Queue* class van vorig week aan zodat je deze voor deze week kunt gebruiken.
4. Schrijf de functie(s) die op de verschillende threads moeten worden uitgevoerd.
5. Test de juiste werking van jouw applicatie door het geluidsbestand te laten bewerken en luister naar het resultaat ! Doordat de berekeningen per *block* worden

uitgevoerd maar het filter afhankelijk is van eerdere samples zullen er wellicht nog kleine 'tikjes' te horen zijn.

### Uitdagingen (voor een cijfer > 8,0):

- Bedenk een manier om de 'tikjes' bij block-grenzen te beperken/voorkomen waarbij nog altijd zo efficiënt mogelijk parallel het geluid gefilterd kan worden en implementeer dit in je programma.
- Maak de applicatie geschikt om met WAV-bestanden te werken i.p.v. pcm raw bestanden.