



FH Salzburg

BACHELORARBEIT 2

Echtzeitorientierung von IntelliSynth mittels Multiprocessing

durchgeführt am Studiengang
Informationstechnik und System-Management
Fachhochschule Salzburg GmbH

vorgelegt von
David Märzendorfer

Studiengangsleiter: FH-Prof. DI Dr. Gerhard Jöchl
Betreuer: Prof. (FH) Univ.-Doz. Mag. Dr. Stefan Wegenkittl

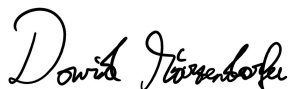
Puch/Salzburg, Januar 2024

Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Weiters versichere ich hiermit, dass ich die benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission weder im In- noch im Ausland vorgelegt und auch nicht veröffentlicht.

29. Januar 2024

A handwritten signature in black ink, reading "David Märzendorfer". The script is cursive and fluid, with the first name "David" and last name "Märzendorfer" clearly distinguishable.

David Märzendorfer, 1910555022

Abstract

IntelliSynth is an experimental setup to create a digital interactive musician. An interactive musician should be able to improvise over a given musical phrase so that it complements a human musician and causes the feeling of creating music together. The aim of this work is to update the original IntelliSynth, which was troubled by an idle-time between the human's input and the playback of the generated improvisation. With the help of Python's *multiprocessing*-module, a solution is formed to get rid of the idle-time. In addition to this, a continuous metronome is implemented. This results in a system which follows the definition of an interactive musician more closely.

Keywords: *Künstliche Intelligenz, MIDI, Transformer, musicautobot, Piano, IPC, Echtzeit*

Inhaltsverzeichnis

1	Einleitung	1
2	Überblick von IntelliSynth v1	2
2.1	Was ist ein interaktiver Musiker	2
2.2	Was ist IntelliSynth	2
2.2.1	Musicautobot	3
2.3	Problematiken in der Ausgangsversion von IntelliSynth	4
3	Echtzeit	6
3.1	Definition	6
3.2	Arten von Echtzeit	6
4	Threads und Prozesse	8
4.1	Definitionen	8
4.2	Synchronisation	10
4.2.1	Semaphore	10
4.2.2	Mutex	10
4.2.3	Scheduler	11
4.3	Interprocess-Communication	11
4.3.1	Shared Memory	11
4.3.2	Signaling	12
4.3.3	Pipe	12
4.3.4	Queue	13
4.3.5	Design Patterns	13
5	Setup und Lösungskonzept	14
5.1	Konzept	14
5.2	Setup Guide	15
5.3	Änderungen von Version 1	17

6	Demo Projekt in Python	18
7	Umsetzung	22
7.1	Metronom	22
7.1.1	Listen	23
7.2	musicautobot	24
8	Zusammenfassung	26
8.1	Ausblick: Latenz	26

Abkürzungsverzeichnis

VM	Virtual Machine
ITS	Informationstechnologie und Systemmanagement
IPC	Interprocess Communication
KI	Künstliche Intelligenz
FIFO	first in, first out
OS	Operating System
I/O	Input/Output
NLP	Natural Language Processing
MIDI	Musical Instrument Digital Interface

Abbildungsverzeichnis

1	Ablauf des gemeinsamen Musizierens. Übernommen aus [1], Seite 1.	2
2	Konstellation der Komponenten. Übernommen aus [1], Seite 2.	3
3	Eine einfache Tokenisierung.	4
4	Tokenisierung, einem Beispiel aus [7] nachempfunden.	4
5	Typen von Echtzeit	7
6	Threads in Prozesse.	9
7	Prozess-Hierarchie.	11
8	Eine Pipe.	12
9	Die IntelliSynth Szenarien.	14
10	Das Lösungskonzept.	15

Tabellenverzeichnis

1	Versionsnummern der Pakete.	17
---	-------------------------------------	----

Quelltextverzeichnis

1	Main Teil-1 der Demo	18
2	Metronom der Demo	19
3	Main Teil-2 der Demo	19
4	Listen der Demo	20
5	Listen der Demo	20
6	Output der Demo	21
7	Das MidiMetronome	22
8	DryPlayback der MidiCapture-Klasse	23
9	Eine verkürzte Version der erweiterten Funktion des musicautobot	24

1 Einleitung

Diese Bachelorarbeit ist eine Weiterentwicklung der Bachelorarbeit 1 *IntelliSynth* [1]. Im weiteren Verlauf dieses Dokumentes wird auf dem bereits beschriebenen Wissen aus *IntelliSynth* aufgebaut. Wichtige Aspekte für das Verständnis der Echtzeitoptimierung werden wiederholt und zusammenfassend erläutert.

Ziel dieser Arbeit ist es, den *IntelliSynth* Prototypen zu verbessern um ein besseres gemeinsames Musizieren zu ermöglichen. Primär sollte dabei die Wartezeit zwischen Einspielen und Prediktion eliminiert werden. Diese Verbesserung wird mit dem Python *multiprocessing*¹ Modul erreicht.

In dieser Arbeit werden folgende Fragen gestellt und beantwortet:

- Wie lässt sich ein interaktiver Musiker gestalten der keine Wartezeit zwischen Zuhören und “eigenem“ Spiel hat und damit das musikalische Metrum nicht stört?
- Wie lässt sich ein solches Konzept mit parallel laufenden Prozessen realisieren?
- Welche Art der Parallelisierung und welche Prozesskommunikation ist dafür geeignet?

¹ <https://docs.python.org/3/library/multiprocessing.html>

2 Überblick von IntelliSynth v1

In diesem Kapitel werden wichtige Aspekte der Bachelorarbeit 1 *IntelliSynth* [1] wiederholt und neu aufbereitet.

2.1 Was ist ein interaktiver Musiker

Ein *interaktiver Musiker* versucht, ein gemeinsames Musizieren zwischen einer Maschine und einem Menschen zu ermöglichen. Er zeichnet sich durch folgende Eigenschaften aus:

- alternierendes Spielen von zwei Parteien
- ein digitales Instrument, in diesem Projekt: ein Digital-Piano
- ein gemeinsames Metrum

Der Ablauf eines gemeinsamen Musizierens kann wie in Abb. 1 aussehen.

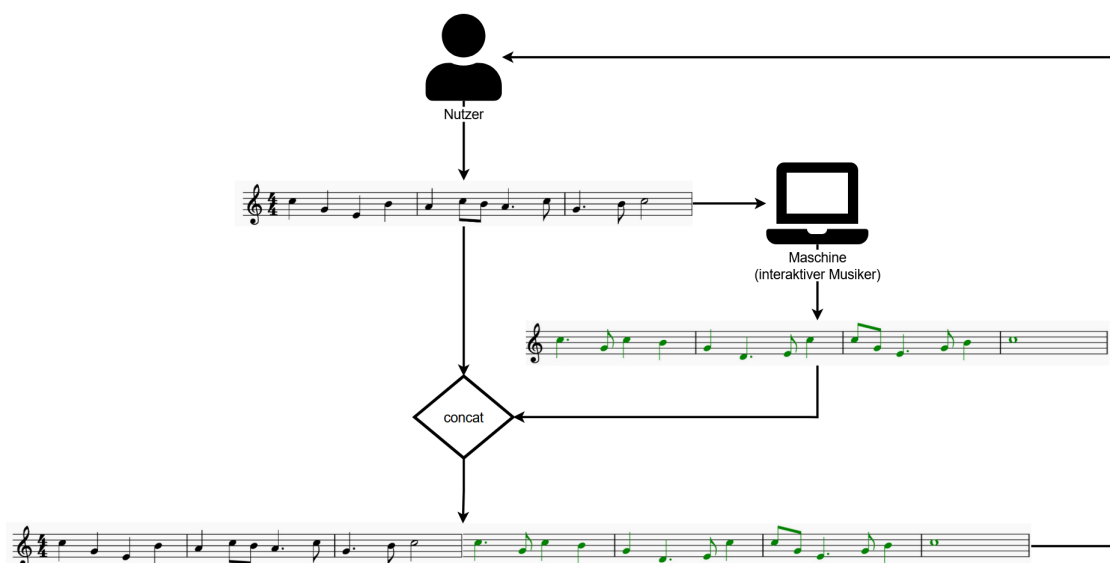


Abbildung 1: Ablauf des gemeinsamen Musizierens. Übernommen aus [1], Seite 1.

2.2 Was ist IntelliSynth

IntelliSynth ist ein System welches versucht einen *interaktiven Musiker* zu ermöglichen. Es besteht aus folgenden Komponenten:

- einem E-Piano

- das *IntelliSynth Programm* welches *musicautobot* [2] zur Erstellung einer Prediktion nutzt
- einem *Zynthian Kit v4* welches die Eingabe und Prediktion als Audio abspielt

Aus den Komponenten ergibt sich folgende Konstellation (Abb. 2):

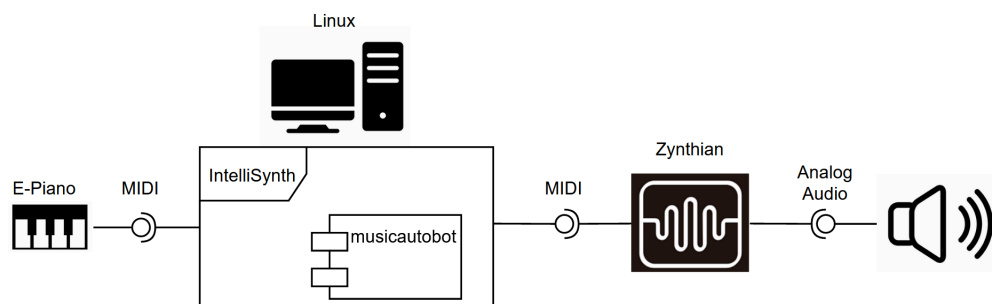


Abbildung 2: Konstellation der Komponenten. Übernommen aus [1], Seite 2.

2.2.1 Musicautobot

Musicautobot [2] ist ein Python Projekt welches *Transformer* nutzt, um Anfangsabschnitte von Musik mithilfe von KI zu vervollständigen oder zumindest zu längeren Stücken zu ergänzen.

Transformer

Transformer [3] stammen aus dem Bereich des *Natural Language Processing (NLP)* [4]. Sie befassen sich also eigentlich mit Texten in verschiedenen Sprachen und deren Übersetzung sowie Vervollständigung. *Musicautobot* verwendet ein *transformerXL*-Modell [5] und wendet dieses auf Musik an.

Bereits in NLP werden Texte als ein *Vokabular* von sogenannten *Tokens* dargestellt. Soll Musiknotation nun vervollständigt werden muss diese auch als *Tokens* in einem *Vokabular* dargestellt werden. Das *Vokabular* stellt alle möglichen *Tokens* dar und bildet somit einen bekannten "Wortschatz". Einzelne Musik-Noten als *Tokens* darzustellen ist relativ einfach möglich. Eine Note hat eine Position auf der chromatischen Tonleiter und eine Dauer. Sie kann also als Kombination von zwei *Tokens* dargestellt werden. Somit ist die *Token*-Darstellung einer F-Halbnote [F:HLF] oder einer H-Viertelnote [H:QRT], wobei die Tokens *HLF* und *QRT* den englischsprachigen Bezeichnungen für Halb- und Viertelnote folgen.

Eine einfache Notenfolge kann wie in Abb. 3 aussehen.

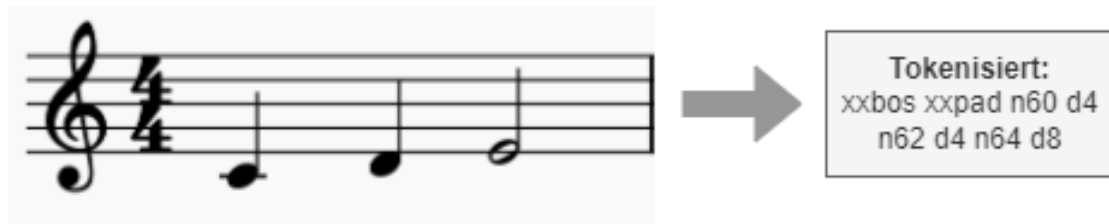


Abbildung 3: Eine einfache Tokenisierung.

Bilden Noten aber einen Akkord, werden sie also zum selben Zeitpunkt gespielt, muss eine Notation gefunden werden, die Gleichzeitigkeit zulässt. Die Lösung dafür sind *Seperator-Tokens* (SEP) welche von dem *bachbot* [6] Modell eingeführt wurden. Diese *SEP-Tokens* werden zwischen Akkorden und einzelnen Noten gesetzt. Diese *SEP-Tokens* haben auch eine Dauer, siehe Abb. 4.

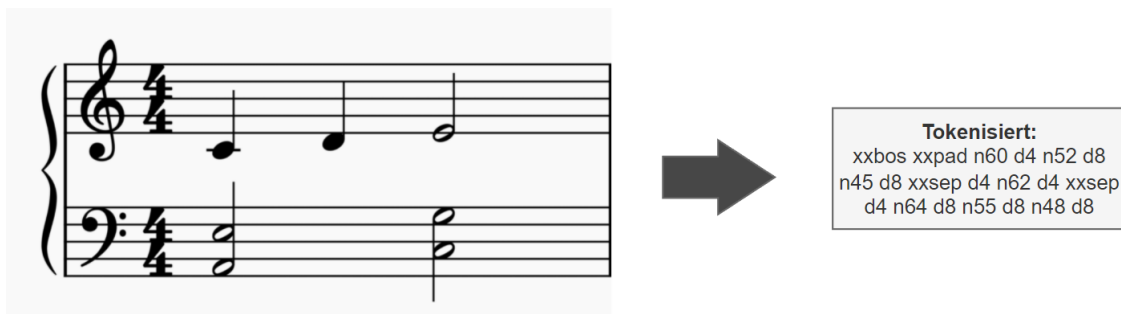


Abbildung 4: Tokenisierung, einem Beispiel aus [7] nachempfunden.

Ein Transformer arbeitet mit diesen *Tokens*. Er erstellt aus anfänglichen Sequenzen, welche in Tokens umgewandelt wurden, eine Sequenz aus Tokens welche rückgewandelt in Noten eine Prediktion oder Vervollständigung der Eingabesequenz darstellen.

2.3 Problematiken in der Ausgangsversion von IntelliSynth

IntelliSynth v1 war bereits in der Lage ein gemeinsames Musizieren mit einer Maschine zu ermöglichen. Es wurde alternierend von Mensch (an einem E-Piano) und der Maschine gespielt. Während der Einspiel-Phase des Menschen ertönt auch ein Metronom, welches sicherstellt, dass die Tokenisierung musikalisch sinnvoll erfolgt. Nachdem die Einspiel-Phase endet (üblicherweise nach vier Takten), stoppt dieses Metronom jedoch und es muss gewartet werden bis die Maschine mit der Prediktion einer Weiterführung fertig, ist damit diese abgespielt werden kann. Während des Abspielen der Prediktion ist kein Metronom mehr vorhanden. Nach der Prediktion wiederholt sich der Prozess. Der Aspekt

des *gemeinsamen Metrum* eines *interaktiven Musikers* ist also nur teilweise abgedeckt. Das Metrum sollte die ganze Zeit spielen. Die Wartezeit stört das Gefühl des gemeinsamen Musizierens. Es soll einen *Echtzeit*-Aspekt geben. Nach dem Einspielen soll sofort die Prediktion der Maschine abgespielt werden.

3 Echtzeit

3.1 Definition

”The main characteristic that distinguishes real-time computing from other types of computation is time.”[8, S. 4]. Diese Zeitkomponente äußert sich in Form von *Deadlines* welche vom System eingehalten werden müssen. Schnelle Systeme werden oft fälschlicherweise als Echtzeit bezeichnet. Die Schnelligkeit eines Systems hat aber nichts mit Echtzeit zu tun. Für die Echtzeit geht es nur darum, ob Aufgaben bis zu einer Deadline erfüllt werden können. Ein Beispiel dafür wäre eine Schildkröte. Eine Schildkröte ist nicht sehr schnell, schafft es aber dennoch wenn Gefahr droht sich *schnell genug* in ihrem Panzer zu verstecken. In ihrer normalen Umgebung ist die Schildkröte also Echtzeitfähig. Ist die Schildkröte einmal zu langsam, kann sie ihre Deadline nicht mehr erfüllen und hat somit das Echtzeit-Szenario versagt.

Betrachtet man nun Echtzeit anhand von *IntelliSynth*: Die Deadline ist das Ende der Einspielphase und die Aufgabe ist das erstellen einer Prediktion, damit diese abgespielt werden kann.

3.2 Arten von Echtzeit

Aufgaben lassen sich in drei Gruppen kategorisieren und somit auch das Echtzeit-System welches diese Aufgaben bearbeitet. Diese Gruppen unterscheiden die Konsequenzen der Nichteinhaltung von Deadlines. [8]

- **Harte Echtzeit:** Eine nicht-Einhaltung einer harten Deadline führt zu katastrophalen Auswirkungen für das System. Ein Beispiel dafür ist das Stabilisierungssystem einer Rakete: kann dieses die gesetzte Deadline nicht erfüllen, stürzt die Rakete ab.
- **Feste Echtzeit:** Wird eine feste Deadline nicht eingehalten, hat das Ergebnis keinen Nutzen mehr für das System. Wird erkannt, dass die Deadline nicht mehr erfüllbar ist wird die Bearbeitung der Aufgabe sofort abgebrochen. Ein Beispiel dafür ist die Dekodierung eines Live-Video-Frames, kann das Frame nicht rechtzeitig dekodiert werden hat es keinen Nutzen mehr und das Frame wird ausgelassen.
- **Weiche Echtzeit:** Wird eine weiche Deadline nicht eingehalten, hat das Ergebnis nur mehr teilweise einen Nutzen für das System und eine Verzögerung hat lediglich Auswirkungen auf die Performance.

Alternativ lassen sich die Gruppen anhand des Nutzens der Aufgabe in Relation zur Zeit und Deadline beschreiben (siehe Abb. 5).

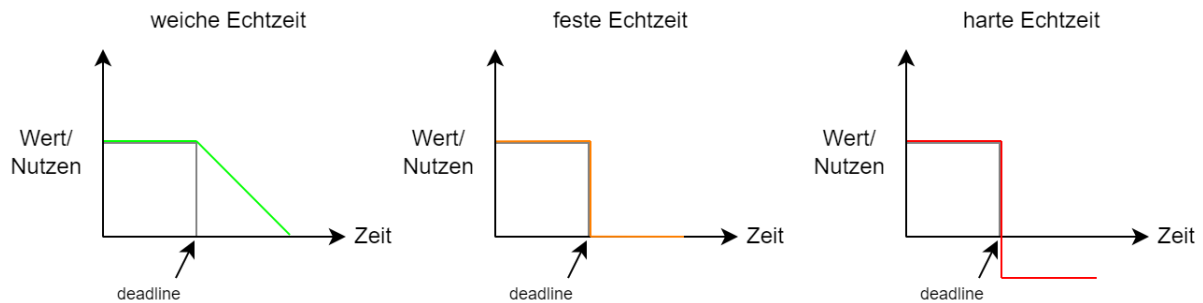


Abbildung 5: *Typen von Echtzeit*

Betrachtet man nun wieder *IntelliSynth*: Endet die Einspielphase bevor eine Prediktion erstellt wurde, muss gewartet werden bis die Prediktion fertig ist. Die Wartezeit führt nicht zu katastrophalen Auswirkungen und die Prediktion hat noch immer einen Nutzen. Die Wartezeit hat negative Auswirkungen auf das gemeinsame Musizieren. Es handelt sich also um eine *weiche Echtzeit*.

4 Threads und Prozesse

4.1 Definitionen

Prozess

Prozesse erlauben es einem Betriebssystem mehrere Aufgaben gleichzeitig zu bearbeiten. So zum Beispiel startet ein Web-Server für jeden Request einen eigenen *Prozess* welcher die Anfrage bearbeitet. Prozesse haben einen eigenen, abgekapselten Adressraum und eine virtuelle CPU. Die CPU ist virtuell, da in einem System mit nur einem CPU-Kern dieser mit mehreren *Prozessen* geteilt werden muss. Ein *Prozess* ist eine Instanz eines laufenden Programms, er hat also die aktuellen Werte des Programmzählers, die Register und Variablen. Zu jeder Zeit kann nur ein *Prozess* mit der CPU arbeiten. Ein *Scheduler* verwaltet, welcher *Prozess* Zugriff auf die CPU hat. In einem System mit mehreren CPU-Kernen können *Prozesse* wirklich parallel arbeiten [9].

Unter *Python's multiprocessing*² gibt es drei Unterscheidungen wie *Prozesse* gestartet werden können: *spawn*, *fork* und *forkserver*.

spawn gibt es auf Unix und Windows. Es wird ein komplett neuer *Prozess* erstellt. Dieser *Process* erhält nur die benötigten Variablen um zu starten. Im Vergleich zu *fork* und *forkserver* ist *spawn* langsamer.

fork gibt es nur auf Unix und ist auch das Standardverhalten von Unix. Hier ist der neu erzeugte Kind-Prozess eine Kopie des Elter-Prozesses. Im Gegensatz zu *spawn* muss ein neuer *Prozess* nicht auf den Stand des *Elter-Prozess* gebracht werden, sondern der *Elter-Prozess* wird einfach kopiert. Nach dem *fork* haben der *Elter-* und *Kind-* Prozess das selbe Speicherabbild, die selben Umgebungszeichenfolgen und die selben geöffneten Daten. Nach dem *kopieren*, kann im neuen *Kind-Prozess* ein anderes Programm als im *Elter-Prozess* laufen [9].

forkserver gibt es auf manchen Unix Systemen. Hierbei gibt es einen *Server-Process* von dem aus alle anderen *Prozesse* geforkt werden.

Neu erstellte *Prozesse* werden *Kind-Prozesse* genannt. Sie haben einen *Elter-Prozess*. Prozesse welche im Hintergrund laufen um bestimmte Aufgaben zu erledigen werden *daemon*

² <https://docs.python.org/3/library/multiprocessing.html>

genannt. Unter Python kann ein Prozess als *daemon* erstellt werden. Dieser Prozess kann keine Kind-Prozesse erstellen, und er wird gestoppt sobald der Haupt-Prozess endet.

Thread

Threads of Execution leben innerhalb von *Prozessen*. Sie teilen sich die CPU und den Adressraum ihres *Prozesses*. Auf einem System kann es also mehrere *Prozesse* geben, welche wiederum mehrere *Threads* besitzen (siehe Abb. 6). *Threads* sind leichter als *Prozesse*, können daher schneller erstellt werden. *Threads* haben einen eigenen *Programmzähler*, welcher die nächsten Instruktionen hält. Weiters haben sie *Register*, welche ihre Variablen halten. *Threads* haben auch einen eigenen *Stack*, welcher die Ausführungshistorie enthält, mit einem Frame für jede Prozedur, die aufgerufen wurde, aber noch nicht zurückgekehrt ist. Obwohl *Threads* in einem *Prozess* leben, werden sie als eigene Konzepte angesehen und können separat betrachtet werden. *Threads* erlauben das Ausführen mehrerer Aufgaben innerhalb eines *Prozesses* [9].

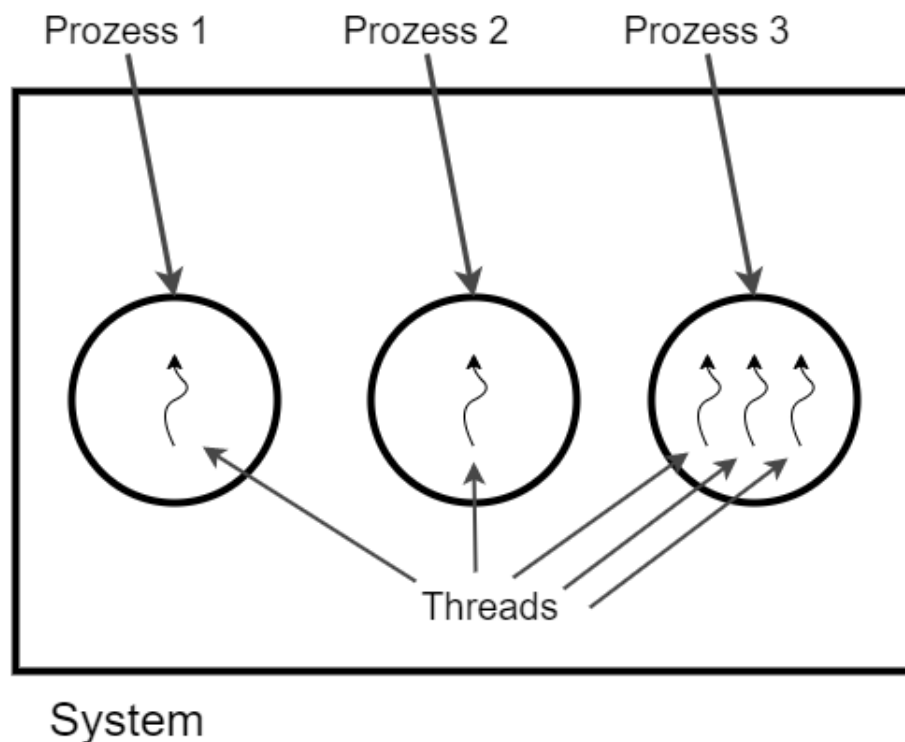


Abbildung 6: *Threads in Prozesse.*

4.2 Synchronisation

Wenn mehrere *Prozesse* auf die selbe *geteilte Ressource* zugreifen, müssen diese synchronisiert werden. Das heißt, der Zugriff muss kontrolliert passieren. Hierfür können unter anderem *Locks* oder *Semaphore* genutzt werden. Folgend wird auf *Semaphore* und *Mutex* genauer eingegangen.

4.2.1 Semaphore

Ein *Semaphore* ist ein einfaches Objekt welches zur Synchronisation genutzt werden kann [10]. Ein *Semaphore* kann als ein Integer mit drei besonderen Eigenschaften beschrieben werden:

- Wir ein neuer *Semaphore* angelegt, wird dieser mit einem positiven Integer Wert initialisiert. Danach kann der Wert des *Semaphore* nur um eins erhöht oder um eins verringert werden. Der aktuelle Wert eines *Semaphore* kann nicht eingesehen werden.
- Wird der Wert von einem Thread/Prozess verringert und der Wert wird negativ, blockiert dieser Thread/Prozess bis der Wert von einem anderen Thread/Prozess erhöht wird.
- Erhöht ein Thread/Prozess den *Semaphore*, wird ein wartender Thread/Prozess aufgeweckt und dieser kann weiter arbeiten.

Ein *Semaphore* Objekt stellt dafür zwei Funktionen zur Verfügung:

- **P:** dekrementiert den Wert um eins.
- **V:** inkrementiert den Wert um eins.

Die Benennung dieser Funktionen stammen von *Edsger Dijkstra*, dem Erfinder der Semaphore.

4.2.2 Mutex

Ein *Mutex* ist ein *mutual exclusive Semaphore*. Er ist ein *Semaphore* mit dem Initialwert Eins. Das heißt, ein *Mutex* kann nur von einem Thread/Prozess gehalten werden. Rufen mehrere Threads auf einen *Mutex* die Funktion **P** auf wird nur Thread A weiterlaufen, der Rest der Threads wird warten bis Thread A den *Mutex* mittels *V* wieder freigibt.

4.2.3 Scheduler

Prozesse müssen sich oft auch *CPU-Leistung* teilen. Welcher *Prozess* wann, wie viel *CPU-Leistung* erhält wird von einem *Scheduler* gesteuert. *Scheduler* agieren auf der *OS-Ebene* zwischen *Process*- und *Hardware-Ebene* (siehe. Abb. 7). Genauere Details zu *Scheduling* und *Synchronisation* sind in [9] zu finden.

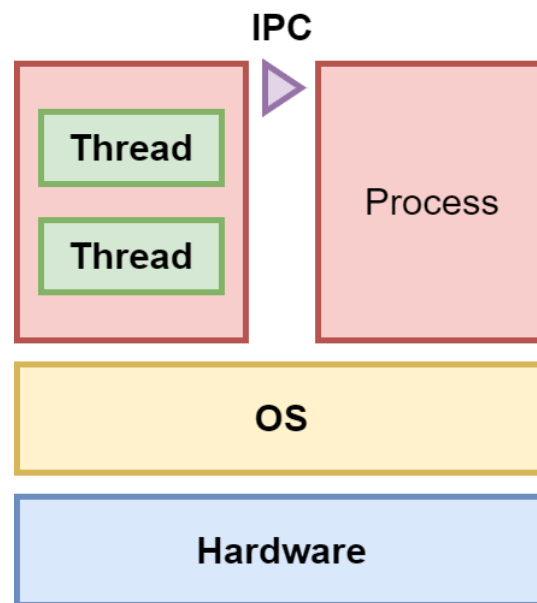


Abbildung 7: Prozess-Hierarchie.

4.3 Interprocess-Communication

Die *Interprocess-Communication* (IPC) beschreibt die Kommunikation zwischen *Prozessen*. Bei dieser Kommunikation muss darauf geachtet werden, dass die *Prozesse* sich nicht gegenseitig in die Quere kommen. Es gibt mehrere Möglichkeiten zur IPC. Hier seien ein paar, für *IntelliSynth* als wichtig angesehen, aufgelistet und erklärt.

4.3.1 Shared Memory

Bei *Shared Memory* haben zwei oder mehr *Prozesse* einen gemeinsamen Speicher auf welchen alle *Prozesse* Schreiben und Lesen können. Die *Prozesse* müssen ihren Lese- und Schreib-Zugriff auf den Speicher synchronisieren um in keine Probleme zu laufen.

4.3.2 Signaling

Signaling ist eine einfache Möglichkeit der IPC. Ein Prozess kann einem anderen Prozess mittels einer *Flag* ein Signal geben. Der zweite Prozess kann auf das Signal entsprechend reagieren. In Python kann hierfür ein *Event* Objekt verwendet werden. Das *Event* Objekt hat vier Funktionen:

- `is_set`: prüft ob die Flag gesetzt ist.
- `set`: setzt die Flag.
- `clear`: löscht die Flag.
- `wait`: blockiert bis die Flag gesetzt ist.

4.3.3 Pipe

Pipes erlauben einen unidirektionalen Informationsaustausch von zwei oder mehr Prozessen [11]. Eine *Pipe* stellt einen FIFO (first in, first out) Speicher mit zwei Enden dar. Ein Prozess schreibt Daten in das eine Ende der *Pipe* und ein zweiter Prozess liest Daten vom anderen Ende der *Pipe*. *Pipes* sind unidirektional, das heißt Daten fließen nur in eine Richtung. Eine Ende der *Pipe* ist immer nur zum Lesen und das andere Ende ist immer nur zum Schreiben (siehe Abb. 8). Soll es einen bidirektionalen Informationsaustausch zweier Prozesse geben, werden zwei *Pipes* benötigt. Prinzipiell können mehrere Prozesse die selbe *Pipe* benutzen. Hierbei muss das Schreiben und Lesen aber synchronisiert und kontrolliert passieren.

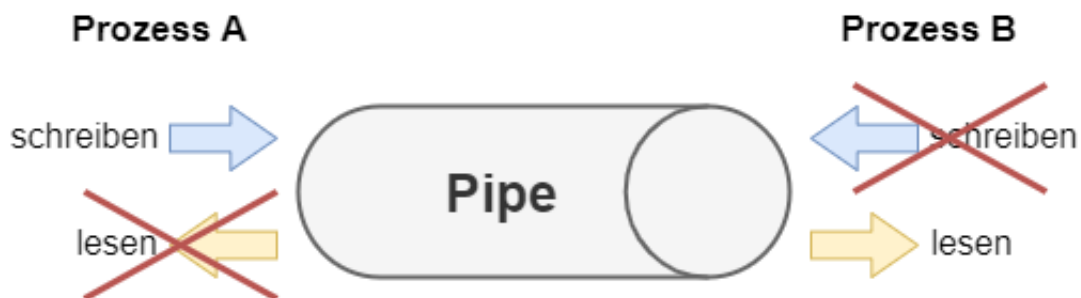


Abbildung 8: Eine Pipe.

Neben normalen *Pipes* gibt es auch *named Pipes*. Diese beschreiben ein selbes Prinzip welches auch für eine Server/Client Verbindung möglich ist.

4.3.4 Queue

Queues sind ähnlich zu *Pipes*. *Queues* sind bidirektional und für mehrere Prozesse ausgelegt. Es kann mehrere gleichzeitig schreibende und lesende Prozesse geben, die Synchronisation wird bereits von der *Queue* gehandhabt.

4.3.5 Design Patterns

Aus der Menge an möglichen Echtzeit Design Patterns wird das folgende näher beleuchtet, da es für dieses Projekt von Relevanz ist. Auf die Implementierung des Pattern wird nicht eingegangen. Nähere Informationen zu Echtzeit Design Patterns sind in [12] zu finden.

Message Queuing Pattern

Das *Message Queuing Pattern* beschreibt eine einfache Möglichkeit für den Informationsaustausch zwischen Threads bzw. Prozessen. Dies sei im weiteren Verlauf anhand von Threads erklärt. Das Pattern besteht aus drei Komponenten: mehreren Threads die kommunizieren wollen, einer Queue und einem geteilten Mutex unter den Threads. Die Queue wird von den Threads verwendet um Informationen zu schicken. Bevor von der Queue gelesen oder geschrieben wird, wird am Mutex **P** aufgerufen. Nachdem Schreiben bzw. Lesen wird der Mutex mit **V** wieder freigegeben. Der Mutex garantiert also, dass zu jeder Zeit nur ein Thread Zugriff auf die Queue hat.

Eine Python Queue aus dem *multiprocessing* Modul hat diesen Locking-Mechanismus bereits eingebaut. Wird nur eine unidirektionale Kommunikation benötigt und gibt es nur einen schreibenden und nur einen lesenden Thread, kann alternativ eine *Pipe* verwendet werden. Eine *Pipe* benötigt keinen Locking-Mechanismus.

5 Setup und Lösungskonzept

5.1 Konzept

Ziel des neuen *IntelliSynth* ist es primär die Wartezeit zwischen Einspielphase und Abspielen der Prediktion zu entfernen. Problematisch hierbei ist jedoch, dass die Prediktion erst beginnen kann sobald der komplette Input aufgenommen wurde. Die Erstellung benötigt aufgrund der verwendeten Algorithmik teilweise mehrere Sekunden und kann daher nicht unbemerkt zwischen Zuhör- und Abspielphase untergebracht werden.

Damit das *Gefühl* einer sofortigen Prediktion erreicht werden kann, muss das Szenario etwas geändert werden. Abb. 9 zeigt den Vergleich zwischen der Ausgangsversion von *IntelliSynth* und dem *gefühlten* Soll-Verhalten.

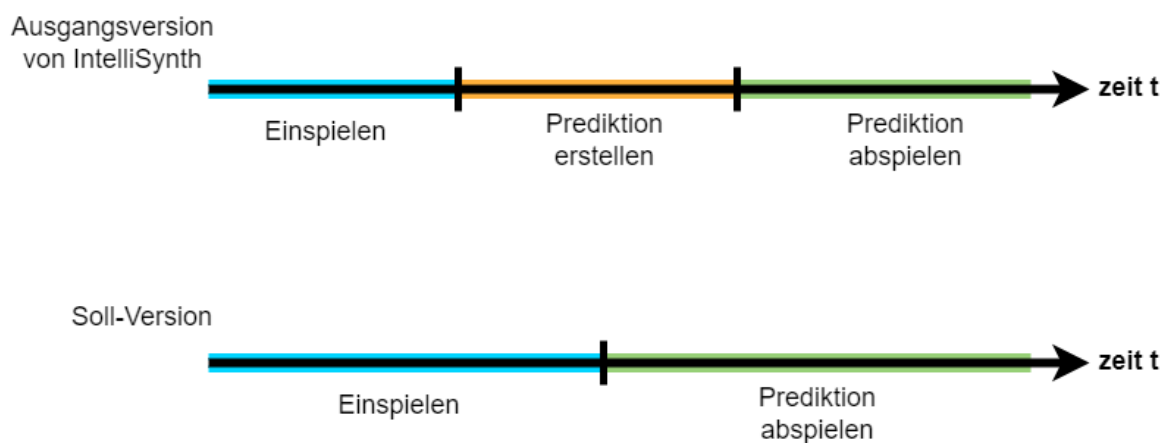


Abbildung 9: Die *IntelliSynth* Szenarien.

Als Lösung wird ein Konzept mit drei *Prozessen* vorgestellt (siehe Abb. 10). Der Zyklus des *Main-Prozesses* besteht aus drei Phasen:

- während der *Einspielphase* wird vom Spieler ein Input aufgenommen.
- während der *Buffer-Phase* kann der Spieler noch weiter spielen, dieser Input wird aber nicht für die Prediktion genutzt
- während der *Abspielphase* wird die Prediktion abgespielt

Parallel zum *Main-Prozesses* gibt es einen *Metronom-Prozesses* welcher durchgehend läuft und dabei ein Signal an den *Main-Prozesses* gibt. Die *Abspielphase* des *Main-Prozesses* startet mit einem Tick.

Zuletzt gibt es noch einen *Prediktions-Prozesses*. Dieser startet nach der *Einspielphase*. Der Prozess schreibt jede generierte Note in eine *Pipe* von der in der *Abspielphase* gelesen wird.

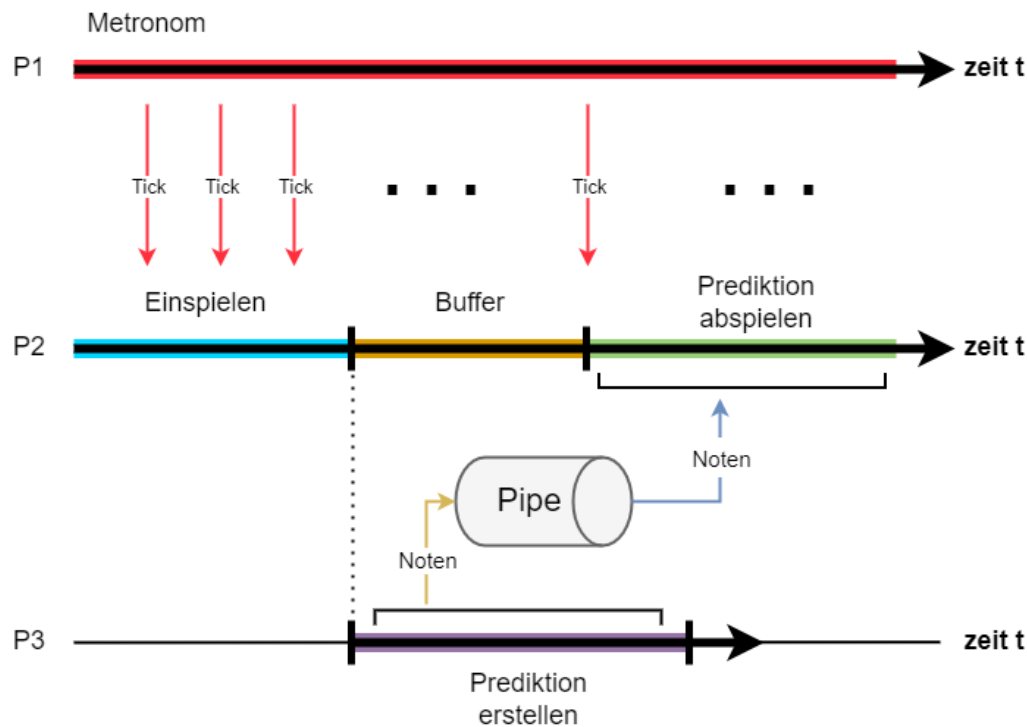


Abbildung 10: Das Lösungskonzept.

Als Zusammenfassung: Für das Lösungskonzept werden *Prozesse* verwendet. Der Prozess P2 ist der Haupt-Prozess. Er kommuniziert mit dem Prozess P1 via *Events* welche von P1 ausgelöst werden. P3 wird mittels eines *fork* aus P2 erstellt. Da zwischen P3 und P2 nur in eine Richtung kommuniziert wird kann eine *Pipe* genutzt werden. Das lesen und schreiben der *Pipe* muss nicht synchronisiert werden, da nur ein Prozess liest bzw. schreibt. Es kann also auf eine *Queue*, welche mehreren Prozesse jeweils synchronisiertes lesen/schreiben erlauben würde, verzichtet werden.

5.2 Setup Guide

Das Setup von *IntelliSynth* hat sich von der Bachelorarbeit 1 [1] nicht stark verändert. Die Schritte für das Setup werden aber dennoch hier wiederholt. Es sei angemerkt, dass das Setup an sich auch auf einer *Unix-VM* möglich ist, aber nicht empfohlen wird. Durch eine *VM* entstehen erhöhte Latenzen im Bereich Input/Output, welche der beschriebenen Optimierung des interaktiven Musikers zuwiderlaufen und vom menschlichen Spieler in

Form von schlechtem Timing wahrgenommen werden.

Schritte für das IntelliSynth Setup

1. Git installieren

```
1 sudo apt install git
```

2. Python installieren

```
1 sudo apt install python3
```

3. Anaconda installieren: <https://www.anaconda.com/products/individual>

4. Pygame installieren

```
1 pip3 install pygame
```

5. libasound installieren

```
1 sudo apt-get install libasound2-dev
```

6. (optional) musescore installieren

```
1 sudo apt-get install musescore3
```

7. das *IntelliSynth* Repository klonen. Der Master-Branch beinhaltet das alte IntelliSynth. Am EchtzeitOptimierung-Branch ist diese Arbeit zu finden.

```
1 git clone git@its-git.fh-salzburg.ac.at:fhs44502/intellisynth.git
```

```
2 git checkout EchtzeitOptimierung
```

8. Als nächstes muss im *IntelliSynth* Ordner der *musicautobot*-Fork³ geklont werden. Dieser Fork beinhaltet geringfügige Änderungen welche für die Echtzeit-Lösung benötigt werden.

```
1 git clone git@github.com:dmaerzendorfer/musicautobot.git
```

9. Gleich wie bei der Bachelorarbeit 1 muss das Setup des *musicautobot*, wie in der README des Projektes beschrieben, durchgeführt werden.

```
1 cd musicautobot
```

```
2
```

```
3 conda env update -f environment.yml
```

```
4
```

```
5 conda activate musicautobot
```

³ <https://github.com/dmaerzendorfer/musicautobot>

10. Final muss noch ein Modell für *musicautobot* mit dem Namen **MultiTaskLarge.pth** in **musicautobot/data/numpy/pretrained/** kopiert werden. Links zu Downloads von Modellen sind im README des *musicautobot* Repositories zu finden.

Für dieses Projekt wurden folgende Versionen der Pakete verwendet:

Paket	Versionsnummer
git	2.36.1
python3	3.10.4
anaconda	2021.11
pygame	2.1.2
libasound2-dev	1.2.6.1
musescore3	3.6.2

Tabelle 1: Versionsnummern der Pakete.

5.3 Änderungen von Version 1

Durch die neue Echtzeit-Version von *IntelliSynth* haben sich folgende Aspekte zum Vorgänger geändert:

- Anstelle eines Metronoms, welches nur während der Einspielphase spielt, gibt es einen durchgehend laufendes Metronom in einem eigenen Prozess.
- das Midi-Modul wurde erweitert, um ein Abspielen ohne Aufzuzeichnen zu erlauben.
- das Aufnehmen wurde geringfügig geändert. Der Spieler hat **n** Beats lange Zeit, etwas einzuspielen. Unabhängig vom tatsächlichen Zeitpunkt des Einsetzens des menschlichen Spielers werden ab diesem Zeitpunkt genau **n** Beats aufgezeichnet.
- *musicautobot* wurde erweitert, um die Prediktion Notenweise in eine Pipe zu schreiben. Das ist nur möglich, da *Transformer* ihre Prediktion wortweise erstellen und somit ein Abgreifen der ersten Noten bereits möglich ist, während die Sequenz selbst noch nicht fertig erstellt ist.
- Integration von Prozessen und IPC zur Synchronisation der erwähnten Bestandteile.

6 Demo Projekt in Python

Bevor mit der wirklichen Umsetzung begonnen wurde, wurde eine vereinfachte Demo des Lösungskonzeptes entwickelt. Die Demo ist im GIT unter `multiprocessingDemo/testMulltiprocessing.py` zu finden. Die Demo stellt das Verhalten der Prozesse und den Ablauf dar, und nutzt keinen Code von *musicautobot* oder dem alten *IntelliSynth*. Folgend wird auf den Code der Demo genauer eingegangen.

```
1  if __name__ == '__main__':
2      #sets how new processes are generated (spawn/fork/forkserver)
3      multiprocessing.set_start_method('spawn')
4
5      #pipes are unidirectional with two endpoints
6      p_output, p_input = Pipe()
7
8      listen_time = 10
9      buffer_time = 5
10     metronome_bpm = 80
11
12     #start the metronome in a process and just let it run the whole time,
13     #everytime it ticks it sends an event
14     tickEvent = Event()
15     metronomeProcess = Process(target=infiniteMetronome, args=(
16         metronome_bpm, tickEvent))
17     print(f"\tstarting the metronome with {metronome_bpm}bpm")
18     metronomeProcess.daemon = True
19     metronomeProcess.start()
20
21     while True:
22         ...
```

Quelltext 1: Main Teil-1 der Demo

Für Pythons multiprocessing wird empfohlen den multiprocessing Code in den `__name__ == '__main__'` Block zu geben. Somit wird der Entry-Point neuer Prozesse abgesichert. Siehe Pythons programming guidelines⁴. Unmittelbar danach wird gesetzt, wie neue Prozesse erstellt werden. Da die Demo in Windows erstellt wurde ist hier `spawn` gesetzt.

Nach dem erstellen einer *Pipe* und einem *Event* wird ein *Metronom-Prozess* erstellt. Dieser Prozess wird als `daemon` konfiguriert.

⁴ <https://docs.python.org/3/library/multiprocessing.html#multiprocessing-programming>

Für das Metronom wird folgender Code verwendet:

```
1
2 def infiniteMetronome(bpm: int, tickEvent: Event):
3     frequency = 440 # Set Frequency To 440 Hertz
4
5     beatDuration = 60 / bpm
6
7     beep_duration = 200 # Set Duration To 200 ms
8
9     while True:
10        #let others know I am about to beep (like a sheep)
11        tickEvent.set()
12        tickEvent.clear()
13
14        winsound.Beep(frequency, beep_duration)
15        #alternatively use the ASCII-Bell, also works on linux
16        #print("\a")
17        time.sleep(beatDuration - beep_duration/1000)
```

Quelltext 2: Metronom der Demo

Das Metronom verursacht im Takt einen Piep-Ton und teilt anderen dies über ein *Event* mit.

```
1 while True:
2     #start to listen
3     print(f"start to listen for {listen_time} seconds")
4     capturedStream = listen(listen_time)
5     #create a predictionProcess which puts its prediction into the pipe
6     predictionProcess = Process(target=predict, args=(capturedStream, (
7         p_output, p_input)))
8
9     #set the process as a daemon, so its stoppen when the main ends
10    #also prevents it from creating processes itself
11    predictionProcess.daemon = True
12    print("\tstart the prediction process")
13    predictionProcess.start()
14
15    #the main process listens for another 10seconds and then starts
16    #playing the prediction
17    print(f"listen for {buffer_time} more seconds")
18    listen(buffer_time)
19    #start the prediction on a metronome tick to assure at least some
20    #synchrony
21    tickEvent.wait()
```

```

19     print("start playing the prediction")
20     while True:
21         #recv() blocks until there is smth to read
22         note = p_output.recv()
23         if note == 'DONE':
24             break
25         else:
26             print(f"Main-Process read: {note}")
27             #pretend to play the note by sleeping 1 second
28             time.sleep(1)
29
30     #once the prediction and its playing is done we restart the cycle
31     #here the process is terminated and a new one is created in the
32     next iteration
33     predictionProcess.join()

```

Quelltext 3: Main Teil-2 der Demo

Im zyklischen Teil der Demo (siehe Quelltext 3) befindet sich die *Einspielphase*, der Start des *Prediktions-Prozesses*, die *Buffer-Phase* und das *Abspielen* der Prediktion.

```

1  def listen(seconds) -> array:
2      capture=[]
3      random.seed(time.time())
4
5      for i in range(seconds):
6          print(f"listening: {i}")
7          capture.append(random.randint(0,100))
8          time.sleep(1)
9
10     return capture

```

Quelltext 4: Listen der Demo

Um die Demo einfacher zu gestalten, wurden die Phasen auf Sekunden statt Beats begrenzt. Die Demo-Version der *Einspielphase* erstellt lediglich jede Sekunde eine Zufallszahl und gibt diese wieder zurück (siehe Quelltext 4). Danach wird ein *Prediktions-Prozess* erstellt und gestartet welcher eine *Pipe* erhält. Der *Prediktions-Prozess* iteriert über die Eingabe. Er multipliziert den Input mit zwei, schreibt das Ergebnis in die Pipe und wartet dann eine Sekunde. Ist der *Prediktions-Prozess* fertig, schreibt er ein "DONE" in die Pipe (siehe Quelltext 5).

```

1  def predict(stream, pipe):
2      p_output, p_input = pipe
3      #start the prediction

```

```
4     for note in stream:
5         #push new notes into the pipe
6         print(f"\tPredicted note: {note * 2}")
7         sys.stdout.flush()
8         p_input.send(note * 2)
9         time.sleep(1)
10
11     #once finished push 'DONE' into the pipe
12     print(f"\tPrediction is complete: sending DONE")
13     sys.stdout.flush()
14     p_input.send('DONE')
15     time.sleep(1)
```

Quelltext 5: *Listen der Demo*

Nachdem der *Prediktions-Prozess* gestartet wurde, wird in der *Main* für weitere fünf Sekunden ein *Buffer-Listen* aufgerufen. Anschließend beginnt die *Abspiel-Phase*. Hier werden die "Noten" aus der Pipe abgespielt, bis ein "DONE" gelesen wird.

Der *Prediktions-Prozess* wird mittels `join` beendet, und ein weiterer Zyklus beginnt.

Ein verkürzter Output der Demo sieht folgendermaßen aus:

```
1         starting the metronome with 80bpm
2 start to listen for 10 seconds
3 listening: 0
4 listening: 1
5 ...
6 listening: 9
7         start the prediction process
8 listen for 5 more seconds
9 listening: 0
10         Predicted note: 92
11 listening: 1
12 ...
13 listening: 4
14         Predicted note: 64
15 start playing the prediction
16 Main-Process read: 92
17         Predicted note: 88
18 ...
19 Main-Process read: 64
20         Prediction is complete: sending DONE
21 Main-Process read: 28
22 ...
```

Quelltext 6: *Output der Demo*

7 Umsetzung

Das echte *IntelliSynth* ähnelt dem Ablauf nach stark der *Multiprocessing-Demo*. Das *Metronom*, das *Listen* (sowie Abspielen) und die *Prediktion* der Demo wurden jedoch ersetzt. Dafür wurden das *MIDI-Modul* sowie *musicautobot* erweitert.

7.1 Metronom

Es wurde unter **midi/MidiMetronome.py** eine neue *MidiMetronome* Klasse erstellt.

Das Metronom bietet zwei Funktionen. Das Ticken für n-Beats und das unendliche Ticken. Beim unendlichen Ticken wird ähnlich wie zur Demo bei jedem Tick ein *Event* ausgelöst (siehe Quelltext 7).

```
1 from multiprocessing import Event
2 import midi.midio.MidiMetadata as MidiMetadata
3 from time import sleep
4
5 class MidiMetronome:
6     __outputDevice = None
7     __channel = 0
8     __bpm = 0
9
10    @classmethod
11    def __init__(cls, outputDevice, bpm, channel):
12        cls.__outputDevice = outputDevice
13        cls.__bpm = bpm
14        cls.__channel = channel
15
16    @classmethod
17    def tickForNBeats(cls, beats):
18        #play the metronome for a certain amount of beats
19        beatDuration = 60 / cls.__bpm
20
21        for beat in range(beats):
22            cls.__sound(beatDuration)
23
24    @classmethod
25    def __sound(cls, beatDuration):
26        #make a single tick sound
27        cls.__outputDevice.playEvent([[MidiMetadata.MidiEvent.noteOn.value
28            , 60, 100, 0], 0], cls.__channel)
29        sleep(0.01)
```

```

29         cls.__outputDevice.playEvent([[[MidiMetadata.MidiEvent.noteOff.
        value, 60, 0, 0], 0]], cls.__channel)
30         sleep(beatDuration - 0.01)
31
32     @classmethod
33     def tickInfinite(cls, tickEvent: Event):
34         #continuously play the metronome
35         beatDuration = 60 / cls.__bpm
36
37         while True:
38             #let others know i will tick now
39             tickEvent.set()
40             tickEvent.clear()
41             cls.__sound(beatDuration)

```

Quelltext 7: *Das MidiMetronome*

7.1.1 Listen

Für das Aufnehmen des Inputs wird weiterhin `createStreamFromLiveInput()` der `MidiCapture` Klasse aus dem alten *IntelliSynth* verwendet. Für den Buffer, wurde jedoch eine neue Funktion `dryPlaybackBeats` erstellt welche lediglich den Input des Spielers an den Zynthian weiterleitet (siehe Quelltext 8).

```

1  @classmethod
2      def dryPlaybackBeats(cls, beats):
3          #for the given amount of beats just output what the player plays
4          duration = beats * (60000000 / cls.bpm) / 1000000
5          cls.dryPlaybackMs(duration)
6
7      @classmethod
8      def dryPlaybackMs(cls, ms):
9          #for the given time just output what the player plays
10         startTime = time.time()
11         #events=[]
12         firstNoteOnReceived = False
13
14         cls.input.flushBuffer()
15         while time.time() <= startTime + ms:
16             event = cls.input.getEvent(1)
17             #we do not want to start playing a note-off event
18             if event is not None and event[0][0][0] == Input.MidiMetadata.
                MidiEvent.noteOn.value:
19                 firstNoteOnReceived = True

```



```

20
21         if event is not None and firstNoteOnReceived:
22             #events.append(event[0])
23             cls.playBack(event, cls.midiChannel)

```

Quelltext 8: *DryPlayback* der *MidiCapture*-Klasse

7.2 musicautobot

musicautobot wurde um eine Prediktion erweitert welche die erstellten Noten in eine *Pipe* schreibt. Dafür wurde ein *git-fork* des *musicautobot*-Repositories erstellt⁵. Es wurde unter **musicautobotmultitask_transformerlearner.py** eine neue Funktion **predict_nw_into_pipe** erstellt. Diese Funktion ist eine Kopie der bestehenden **predict_nw** Funktion, mit dem Unterschied, dass die Prediktion auch in eine *Pipe* geschrieben wird. Auf die Funktionsweise der Prediktion wird nicht genauer eingegangen, es wird lediglich die Erweiterung durch eine *Pipe* erklärt.

Transformer erstellen ihre Prediktion wortweise. *Musicautobot* arbeitet mit Musik, die Noten werden durch zwei Token dargestellt (Note und Dauer). Gleichzeitig gespielte Noten werden durch ein *SEP*-Token gekennzeichnet (siehe 2.2.1). Damit nur komplette Noten in die *Pipe* geschrieben werden, darf erst nach einem *SEP*-Token agiert werden. Hierbei ist wichtig zu beachten, dass *SEP*-Tokens auch eine Dauer haben. Dafür wurde ein **last_note** Array angelegt welches die noch nicht geschriebenen Tokens beinhaltet. Sobald ein *SEP*-Token und eine Dauer prognostiziert werden, werden die Tokens in **last_note** in ein *MusicItem* umgewandelt und in die *Pipe* geschrieben (siehe Zeile 33ff im Quelltext 9). Am Ende der Prediktion werden die übrigen Tokens als *MusicItem* in die *Pipe* geschrieben.

```

1  def predict_nw_into_pipe(self, pipe:Pipe, item:MusicItem, n_words:int=128,
2      temperatures:float=(1.0,1.0), minBars=4,
3      top_k=30, top_p=0.6):
4      "Return the `n_words` that come after `text`. Whilst also putting
        the tokens into a pipe"
5      p_output, p_input = pipe
6
7      self.model.reset()
8      new_idx = []
9      last_note = []
10     vocab = self.data.vocab
11     x, pos = item.to_tensor(), item.get_pos_tensor()
12     last_pos = pos[-1] if len(pos) else 0

```

⁵ <https://github.com/dmaerzendorfer/musicautobot>

```
13         y = torch.tensor([0])
14
15         start_pos = last_pos
16
17         sep_count = 0
18         bar_len = SAMPLE_FREQ * 4 # assuming 4/4 time
19         vocab = self.data.vocab
20
21         repeat_count = 0
22
23         for i in progress_bar(range(n_words), leave=True):
24             batch = { 'lm': { 'x': x[None], 'pos': pos[None] } }, y
25             logits = self.pred_batch(batch=batch)['lm'][-1][-1]
26
27             prev_idx = new_idx[-1] if len(new_idx) else vocab.pad_idx
28
29             #transformer-magic
30             #...
31             #####
32
33             new_idx.append(idx)
34             last_note.append(idx)
35
36             #if a note is complete->put it into the pipe
37             if prev_idx == vocab.sep_idx:
38                 arr2 = np.array(last_note)
39                 arr = np.array(new_idx)
40
41                 item = vocab.to_music_item(arr2)
42                 p_input.send(item)
43                 last_note = []
44
45                 x = x.new_tensor([idx])
46                 pos = pos.new_tensor([last_pos])
47
48             pred = vocab.to_music_item(np.array(new_idx))
49             full = item.append(pred)
50             #send last music_item into the pipe
51             p_input.send(vocab.to_music_item(np.array(last_note)))
52
53         return pred, full
```

Quelltext 9: Eine verkürzte Version der erweiterten Funktion des *musicautobot*

8 Zusammenfassung

In Folge der Arbeit wurde *IntelliSynth* dahingehen optimiert Wartezeiten zu eliminieren. Nach den Verbesserungen gibt es ein durchgehend laufendes *Metronom* und es spielt zu jederzeit entweder der Mensch oder die Maschine. Die Eliminierung der Wartezeit wurde durch *multiprocessing* erreicht. Prozess-Kommunikation wird über *Events* und *Pipes* gewährleistet.

Durch diese Änderungen ist *IntelliSynth* einem interaktiven Musiker etwas näher gekommen. Es besteht aber noch das Problem der *Latenzen*.

8.1 Ausblick: Latenz

Besonders bei der Verwendung von *IntelliSynth* auf einer VM sind die Latenzen des Systems sehr gravierend. Zwischen Tastendruck am E-Piano und dem Abspielen des Tons besteht eine Latenz. Bevor das Signal der Taste zum Zynthian-Kit gelangt um abgespielt zu werden, wird das Signal über *IntelliSynth* aufgezeichnet und wird dann erst weitergeleitet. Durch diese Schnittstellen-Wechsel (besonders in einer VM) entsteht eine hohe Latenz. Die Verzögerung zwischen Tastendruck und abspielen eines Tons schadet dem Musizieren.

Eine einfache Lösung dafür wäre, lediglich die Tastendrucke Aufzunehmen und nicht abzuspielen. Das Abspielen kann direkt über das E-Piano gehandhabt werden, hier besteht weniger Latenz.

Literaturverzeichnis

- [1] F. Hinterberger und D. Märzendorfer, *IntelliSynth*, Bachelorarbeit. Fachhochschule Salzburg, 2022.
- [2] bearpelican, *musicautobot*, <https://github.com/bearpelican/musicautobot>, [Online; accessed 2021-11-20].
- [3] A. Vaswani u. a., »Attention is All you Need,« in *Advances in Neural Information Processing Systems*, I. Guyon u. a., Hrsg., Bd. 30, Curran Associates, Inc., 2017.
- [4] D. Jurafsky und J. H. Martin, *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*. Upper Saddle River, N.J.: Pearson Prentice Hall, 2009, ISBN: 9780131873216.
- [5] Z. Dai u. a., »Transformer-XL: Attentive Language Models beyond a Fixed-Length Context in Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics,« (Florence, Italy), Jan. 2019, S. 2978–2988. DOI: 10.18653/v1/P19-1285.
- [6] F. T. Liang u. a., »Automatic Stylistic Composition of Bach Chorales with Deep LSTM,« in *18th International Society for Music Information Retrieval (ISMIR)*, (Suzhou, China, 23.–27. Okt. 2017), S. 449–456.
- [7] A. Shaw. »Practical Tips for Training a Music Model.« [Online; accessed 2021-12-11]. (Aug. 2019), Adresse: <https://towardsdatascience.com/practical-tips-for-training-a-music-model-755c62560ec2>.
- [8] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd. Springer Publishing Company, Incorporated, 2011, ISBN: 1461406757.
- [9] A. S. Tanenbaum und H. Bos, *Modern Operating Systems*, 4th. USA: Prentice Hall Press, 2014, ISBN: 013359162X.
- [10] A. B. Downey, *The Little Book of SEMAPHORES (2nd Edition): The Ins and Outs of Concurrency Control and Common Mistakes*. USA: Green Tea Press, 2016, ISBN: 1441418687.
- [11] J. Gray, *Interprocess Communications in Linux*. Prentice Hall PTR, 2003, ISBN: 9780130460424. Adresse: <https://books.google.at/books?id=Y7hQAAAAMAAJ>.
- [12] B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0201699567.