



Evangelisches Gymnasium Werther

Facharbeit

Prozesse und Threads

Anhand einer Bild Software

Verfasser: **Philipp Bleimund**

Kurs: Grundkurs Informatik

Betreuende Lehrkraft: Herr Christian Möllenbrock

Abgabetermin: unknown

Inhaltsverzeichnis

| | |
|--|-----------|
| 1 Threads und Prozesse | 3 |
| 1.1 Einleitung | 3 |
| 1.2 Aufbau von Prozessen | 3 |
| 1.3 Verwalten der Prozesse | 5 |
| 1.4 Funktionsweise des Schedulers | 7 |
| 1.4.1 Completely Fair Scheduling | 9 |
| 1.4.2 Round Robin | 11 |
| 1.5 Threads | 11 |
| 1.6 Threads in Java | 12 |
| 1.6.1 Implementation in Java | 12 |
| 1.6.2 ThreadPool | 13 |
| 1.6.3 Thread sicherheit | 13 |
| 2 Quellen | 13 |
| 3 Programm | 14 |
| 3.1 Einleitung | 14 |
| 3.2 Idee | 15 |
| 3.3 Implementation in Java | 15 |
| 3.3.1 Berechnen der Sektionen | 15 |
| 3.3.2 Berechnen der durchschnittlichen Farbe | 16 |
| 3.3.3 Erstellen einer Datenbank | 17 |
| 3.3.4 Berechnen der besten Bilder | 18 |
| 3.3.5 Skalieren der Bilder | 19 |
| 4 Laufzeitanalyse | 24 |
| 4.1 Einleitung | 24 |
| 4.2 Testungen | 25 |

1 Threads und Prozesse

1.1 Einleitung

Prozesse sind die Ausführung eines Programms auf dem Prozessor. Jedoch kann ein Prozessor maximal ein Prozess gleichzeitig ausführen. Um Verwirrung zu beseitigen möchte ich darauf hinweisen, dass selbst moderne Prozessoren nicht in der Lage sind mehrere Prozesse auszuführen. Diese "Illusion" wird erzeugt, da ein Prozessor(Bauteil) mehrere Kerne hat. Diese Kerne sind die eigentlichen Prozessoren. In Zukunft werde ich den Begriff Kerne nutzen um die Unterscheidung zu erleichtern. Um trotzdem mehrere Prozesse gleichzeitig zu bearbeiten, werden den einzelnen Kernen die Prozesse für nur wenige Millisekunden zugeordnet. Diese nennt man auch Virtuelle Threads. Jeder Virtuelle Thread kann einem realen Kern zugeordnet werden. Jedoch wird nicht jeder Process gleich lange einem Kern zugeordnet. Die Prozesse konkurrieren um ihre Zeit. Denn schließlich soll mein Computerspiel nicht die gleiche Zeit bekommen, wie meine Stoppuhr app. Diese wechselt zwischen den einzelnen Prozessen nennt man auch Kontextwechsel. Der Kontext des Kerns ändert sich demnach.

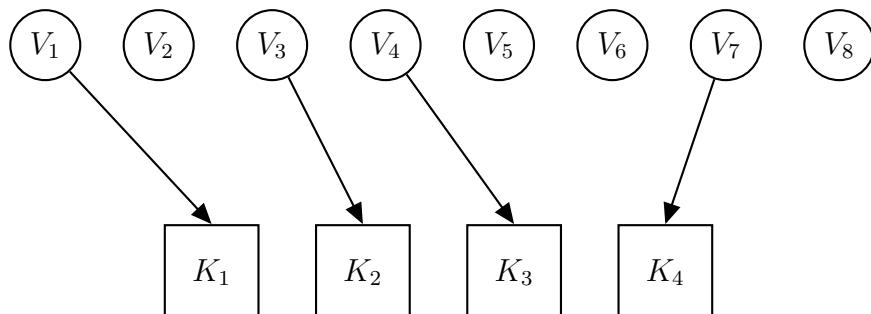


Abbildung 1.1: Aufteilung von Virtuellen Threads auf Kernel

1.2 Aufbau von Prozessen

Prozesse müssen jedoch noch ein wenig mehr als nur ein Stück code besitzen, um aktiv zu werden. Generell kann man sagen, dass Prozesse aus 7 Elementen bestehen. Diese nennt man Prozesskontext. Innerhalb des Prozesskontextes gibt es noch den Hardwarekontext.

- Das auszuführende Programm
- Die Daten des Programmes. Umfasst etwa die Globalen Variablen.
- Einen Stack. Ein Stack funktioniert nach dem Push und Pop Verfahren und speichert die lokalen Variablen für einen schnelleren Zugriff.

- Kernelstack. Umfasst die Systemaufrufe des Prozesses.
 - CPU Register. Kann in den meisten Fällen nur ein Befehl speichern (64bit Prozessor = 64bits im Register)
 - MMU Register, dass den Zugriff auf den Arbeitsspeicher verwaltet.

Da ein Prozess viele Kontextwechsel durchleben wird, muss das Betriebssystem bestimmte Register speichern. Dazu gehört aus dem Hardwarekontext:

- Instruction Pointer - Die Speicheradresse des nächsten Befehls.
- Instruction Register - Der aktuelle Befehl.
- Stackpointer - Speichert das Ende des Stacks.
- Basepointer - Speicheradresse des aktuellen Elementes im Stack.
- Akkumulator - Speichert, Ergebnisse der ALU

Dies sind die wichtigsten Informationen, um die Rechenoperationen weiterführen zu können. Das Betriebssystem braucht jedoch noch weitere Informationen über einen Prozess. Diese werden auch Systemkontext genannt. Die wichtigsten davon sind:

- Ort in der Prozesstabellen
- PID - Prozessnummer
- Prozesszustand
- Priorität
- Eltern- oder Kindprozesse
- Zugriffsrechte - Linux: -20 bis 19; Windows: Rechte werden einzeln zugewiesen
- Erlaubte Ressourcenmengen - Bsp. Maximaler RAM Verbrauch
- Verwendete Dateien - Um zu verhindern, dass mehrere Prozesse an einer Datei arbeiten
- Zugeordnete Geräte - Maus, Tastatur, ...

Mithilfe der Prozesstabellen kann das Betriebssystem die einzelnen Prozesse speichern. In dieser werden Prozesskontrollblöcke gespeichert, welche den Hardwarekontext und Systemkontext beinhaltet. Bei einem Kontextwechsel wird der Prozesskontext aus der Prozesstabellen wieder hergestellt.

1.3 Verwalten der Prozesse

Jedes Betriebssystem muss einen weg haben, um effektiv die Kontextwechsel der Prozesse durchführen zu können. Dazu wird in den meisten Fällen ein *Warteschlangen System* verwendet. Auch hat ein Prozess deutlich mehr Zustände als nur *untätig* und *rechnend* in einem modernen Betriebssystem. Dazu wird heutzutage meistens das *7-Zustands Prozessmodell* oder eine modifizierte Variante. Linux als Beispiel verwendet ein *8-Zustands Prozessmodell*, welches das Modell mit einem *kernel rechnend* Zustand erweitert.

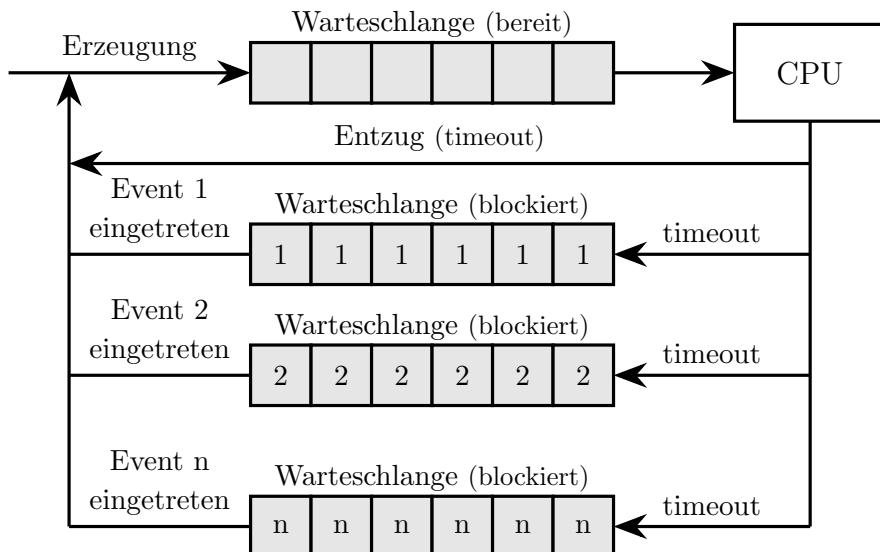


Abbildung 1.2: Warteschlangen System

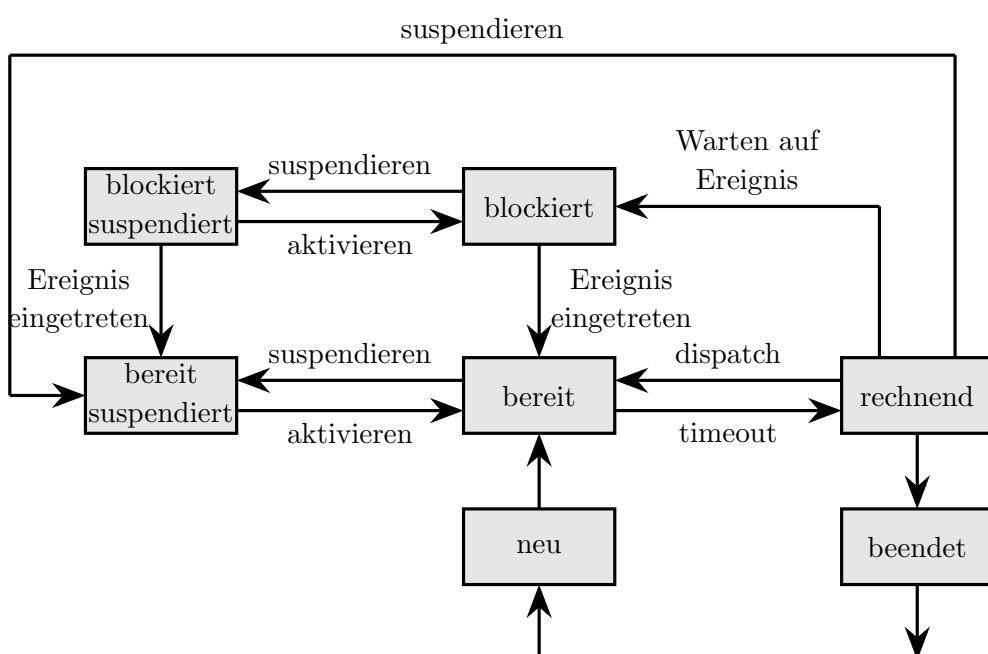


Abbildung 1.3: 7-Zustands Prozessmodell

Wie in der Einleitung schon angesprochen sind die Zustände *bereit* und *rechnend* die wichtigsten Zustände. Mit diesen alleine könnte ein Betriebssystem funktionieren. Es gäbe dazu dann nur eine Warteschlange, in der sich alle Prozesse des Zustandes *bereit* befinden. Idealer Weise implementiert der *Scheduler*¹ einen Algorithmus, welcher die Priorität der Prozesse berücksichtigt. Wie schon erwähnt muss sich der *Dispatcher*² um noch weitere Zustände kümmern. Diese und ihre Beziehungen sind in Grafik 1.3 zu finden. Zwei davon währen *neu* und *beendet*. Diese sind für eine größere Flexibilität nützlich. Mit dem *beendet* Zustand, können Informationen nachträglich von einem fertigen Prozess aufgerufen werden. Der Zustand *neu* hat die gemeinsame Funktion mit dem *beendet*-Zustand Ressourcen zu sparen.

Ein Entschiedener Fehler ist es anzunehmen, dass alle Prozesse jederzeit Arbeiten wollen. So könnte ein Programm auf eine Tastatur Eingabe oder andere Ereignisse warten. Um diese Funktionalität bereitstellen zu können gibt es den Zustand *blockiert*. In diesen wechselt ein Prozess nach den berechnen und kann aus diesen sich wieder in die Warteschlange der bereiten Prozesse einordnen. In Grafik 1.2 werden unterschiedliche Warteschlangen für unterschiedliche Ereignisse erzeugt. Dieses Vorgehen hat den Vorteil gegenüber einer einzelnen “*blockiert*-Warteschlange”, dass häufig genutzte Events wie Tastenanschläge nicht von seltenen Events beeinträchtigt werden.

Da es sehr schnell zu vielen Prozessen kommen kann, wird mit den Zuständen *blockiert suspendiert* und *bereit suspendiert* eine Möglichkeit geschaffen, selten genutzte Prozesse aus dem Arbeitsspeicher in den Massenspeicher³ zu verschieben. Wie die Namen schon Implizieren Prozesse in den Zuständen *blockiert* und *bereit* jeweils suspendiert und aktiviert werden. Für zusätzliche Geschwindigkeit, können Prozesse selbst im suspendierten Zustand auf Ereignisse reagieren und von *blockiert suspendiert* in *bereit suspendiert* wechseln. Es gibt demnach ein zweites *Warteschlangen System* für die suspendierten Prozesse. Dieses beinhaltet keinen zugriff auf die CPU, sondern kann die Prozesse maximal aktivieren und in den Arbeitsspeicher verschieben. [1]

¹ Programm zum Managen der Warteschlangen.

² Programm zum ausführen der Prozesswechsel.

³ Spezielle Partitionen auf einer Festplatte. Auch *swap* genannt.

1.4 Funktionsweise des Schedulers

Der *Scheduler* ist ein sehr wichtiges und mächtiges Stück Code. Es manages alle anderen Prozesse eines Betriebssystems. Es kann sich die Frage gestellt werden, wie der *Scheduler* ausgeführt wird. Ist er nur ein weiterer Prozess? Dies würde aber implizieren, dass er sich selber managen würde. Oder wird er auf einem eigenen CPU Kern ausgeführt? Aber Linux läuft doch auch auf einem einzelnen Kern. Die Antwort liegt in der Natur des Kernels.

Der Kernel ist die niedrigste Instanz mit der höchsten Berechtigung in einem System. Nichts steht neben ihm und der CPU. Jedes Programm muss über den Kernel um etwas machen zu können. Der *Scheduler* ist ein Teil des Kernels. Der Kernel ist jedoch kein einzelner Prozess, welcher immer läuft, sondern eine Art Bibliothek. Ein Programm wendet sich an den Kernel und nicht der Kernel an das Programm. Dementsprechend läuft der *Scheduler* nicht dauerhaft, sondern wird nach einer gewissen Zeit getriggert. Der *Scheduler* wird dabei entweder von einem beendeten Process getriggert oder nach einer Zeitunterbrechung. Die Zeitunterbrechung wird dabei von der CPU durch den *programmable interrupt timer (PIT)* erzeugt und im Kernel durch den *timer interrupt handler* aufgefangen, welcher auch den *Scheduler* startet. Es wird dabei zwischen einem *ticked kernel* und *tickless kernel* unterschieden. Bei dem *ticked kernel* ist der Zeitintervall immer gleich wogegen der des *tickless kernels* dynamisch verändert werden kann.[6]

Der *Scheduler* verwaltet die einzelnen Warteschlangen und entscheidet, wann ein Prozess auf die CPU zugreifen darf. Dabei ist es wichtig die beste Effizienz beizubehalten und trotzdem eine gute Verteilung der Prozesszeit zu ermöglichen. Denn Kontextwechsel sind aufwändig. Bei vielen kleinen Prozessen wird viel Zeit für das speichern der Register und das wiederherstellen eines Prozesses aus der *Prozesstabellen* verwendet. Je länger ein Prozess arbeiten kann, desto effizienter wird die Zeit genutzt. Daher haben sich zwei grundlegende Konzepte des *schedulings* gebildet. Diese Schedulingverfahren sind:

- *Nicht-präemptives Scheduling*. Bei diesem ist ein Prozess bis zu seiner Fertigstellung über volle Kontrolle über die CPU. Der *Scheduler* führt erst den Kontextwechsel nach dessen Vollendung aus. Dabei kann es zu Situationen kommen, bei denen ein Prozess sich nicht beendet. Beispielsweise durch eine Endlosschleife in der Programmierung oder der Entwickler setzt bei der Programmierung das *Präemptive Scheduling* voraus.
- *Präemptives Scheduling*. Dieses wird seit Windows 3.x und Mac OS8/9 verwendet. Dabei wird nicht auf die Vollendung eines Prozesses gewartet, sondern kann , und wird in den meisten Fällen, der Prozess der

CPU vor Beendung entzogen. Der Vorteil ist, dass viele weitere Prozesse „gleichzeitig“ arbeiten können, ohne dass der Nutzer das „Einfrieren“ anderer Prozesse erfährt. Der Nachteil dabei ist, dass die Kontextwechsel viel Zeit in Anspruch nehmen. Somit haben die Prozesse weniger Arbeitszeit und die gesamte Leistung der CPU sinkt etwas. Da die Vorteile der größeren Freiheit der leicht verringerten Leistung überwiegen, wird dieses Verfahren in den meisten modernen Betriebssystemen verwendet.

[1]

Auf diesen zwei grundlegenden Systemen haben sich weitere *Scheduling* Verfahren entwickelt, welche das Verwalten der Warteschlange implementieren. Um ein Ideales System zu erschaffen, müssen bestimmte Punkte berücksichtigt werden. Diese lassen sich nicht immer miteinander vereinbaren und es ist dem Entwickler überlassen, welche er bevorzugt. Diese Kriterien währen wie folgt: [11]

- Prozessor-Auslastung - Die Prozessor-Auslastung sollte im Idealfall so hoch wie möglich sein. Damit kein Befehlszyklus¹ verschwendet wird.
- Antwortzeit - Die Zeit, die vergeht, bis die erste Antwort eines Prozesses nach Anfrage ankommt.
- Durchlaufzeit - Die Zeit, die vergeht, bis ein Prozess nach Einreichung beendet ist.
- Durchsatz - Wie viele Prozesse in einem vorgegebenen Intervall, beendet werden. Der Intervall kann je nach Anwendungsfall variieren.
- Wartezeit - Die Zeit, die ein Prozess in der bereit-Warteschlange verbringt, bis er Zugriff auf die CPU bekommt.
- Fairness - Die Fairness eines Verfahrens bestimmt, wie gut kleine und weniger priorisierte Prozesse eine Chance haben Prozesszeit zu erhalten.

Im Folgenden werde ich zwei *Scheduling* Verfahren vorstellen.

¹ Ein Befehlszyklus ist der kleinste Zeitintervall einer CPU. Befehle können unterschiedlich viele Befehlszyklen brauchen. Je kleiner der Befehlszyklus ist desto höher ist die Herz Anzahl einer CPU

1.4.1 Completely Fair Scheduling

*Completely Fair Scheduling*¹ ist eine, wie der Name schon impliziert, eine Form des *Ideal Fair Scheduling*. Dieses besagt, dass versucht wird die Prozesse gleich lange Arbeiten zu lassen. In den Tabellen 1.1 sind vier Prozesse dargestellt. Alle haben die selbe Priorität. In dem Szenario hat jeder Zeit-Quant (Q_1, Q_2, \dots) eine Zeitspanne von 4ms. In den ersten vier Quanten bekommt jeder Prozess 1ms. Ab Q_4 sind Prozess B und D abgeschlossen und A und C erhalten somit in Q_5 jeweils 2ms. Durch das Prinzip werden alle Prozesse gleich behandelt. [8]

| Prozess | Ausführungszeit | | Q_1 | Q_2 | Q_3 | Q_4 | Q_5 | Q_6 | Q_7 | Q_8 |
|---------|-----------------|---|-------|-------|-------|-------|-------|-------|-------|-------|
| A | 8ms | A | 1 | 2 | 3 | 4 | 6 | 8 | | |
| B | 4ms | B | 1 | 2 | 3 | 4 | | | | |
| C | 16ms | C | 1 | 2 | 3 | 4 | 6 | 8 | 12 | 16 |
| D | 4ms | D | 1 | 2 | 3 | 4 | | | | |

Tabelle 1.1: *Fair Scheduling (FS)*

Im CFS wird das Prinzip des FS nicht hauptsächlich mit Zeitscheiben gelöst. Es wird das Konzept der *vruntime* eingebracht. *Vruntime* bedeutet dabei, wie lange ein Prozessor bereits ausgeführt wurde. CFS berechnet weiterhin unterschiedliche Zeitscheiben für die PIT, jedoch werden diese zur Maximierung der Effizienz genutzt. Prozesse mit komplexen Rechnungen sind effizienter, wenn sie mehr Zeit ohne Kontextwechsel haben. Andere Prozesse wie Tracker schadet der Kontextwechsel nicht allzu sehr. Der *Scheduler* addiert auf die alte *vruntime* die Zeit(Δt) die der Processor lief. Sollte der Prozess noch nicht abgeschlossen sein, wird die neue *vruntime* in einen Rot-Schwarz Baum eingefügt. Ein Rot-Schwarz Baum ist ein Binärbaum, welcher eine selbst ausgleichende Natur besitzt. Somit beträgt die einfache Laufzeit $\mathcal{O}(\log n)$. CFS wählt als nächsten Prozess, den mit der geringsten *vruntime* aus. Dabei wird immer ein Pointer auf das kleinste Blatt gehalten, um eine Laufzeit von $\mathcal{O}(1)$ zu erhalten, um den Prozess mit der kleinsten *vruntime* zu finden. Durch das wählen der kleinsten *vruntime* wird das Prinzip des *Fair Scheduling* eingehalten. [8]

¹ *Completely Fair Scheduling (CFS)* wurde 2007 das erste mal von Ingo Molnar im Linux Kernel eingearbeitet.

Das Nutzen der *vruntime* als zentralen Wert hat weitere Vorteile. So können die Prioritäten ohne Umwege eingearbeitet werden. Der CFS verwaltet in der *vruntime* drei unterschiedliche *Scheduling Policies*. [9]

- **SCHED_NORMAL/SCHED_OTHER.** Die normale Regel für Prozesse.
- **SCHED_BATCH.** Für Prozesse, die nicht interaktiv sind und den Arbeitsbereich des Nutzers nicht stören wollen.
- **SCHED_IDLE.** Die niedrigste Regel für Prozesse und sie werden an wenigsten bevorzugt.

SCHED_NORMAL und SCHED_BATCH sind zusätzlich von den *nice* Werten abhängig. Die *nice* Werte reichen von -20 bis 19 und repräsentieren die Priorität eines Prozesses. Je kleiner der *nice* Wert, desto höher ist die Priorität[5]. Dabei ist der Unterschied Linear. Die Verteilung des Prozessors auf zwei Prozesse mit jeweils 11 und 12, wäre 55% und 45%. Bei Prozessen mit -4 und -5 ist auch 55% und 45%. Die neue *vruntime* wird mit Formel 1.1[10] berechnet.

$$\text{vruntime} += \text{delta_exec} \cdot \frac{\text{weight}}{\text{lw.weight}} \quad (1.1)$$

Dabei ist *weight* mit 1024 oder *nice_0* definiert. *lw.weight* ist die Priorität als *weight*. Dieses kann aus dem Array in dem Listing 1.1[10] entnommen werden.

```

10886 const int sched_prio_to_weight[40] = {
10887 /* -20 */ 88761, 71755, 56483, 46273, 36291,
10888 /* -15 */ 29154, 23254, 18705, 14949, 11916,
10889 /* -10 */ 9548, 7620, 6100, 4904, 3906,
10890 /* -5 */ 3121, 2501, 1991, 1586, 1277,
10891 /* 0 */ 1024, 820, 655, 526, 423,
10892 /* 5 */ 335, 272, 215, 172, 137,
10893 /* 10 */ 110, 87, 70, 56, 45,
10894 /* 15 */ 36, 29, 23, 18, 15,
10895 };

```

Listing 1.1: Linuxkernel core.c(v5.17-rc3) *nice* Werte als *weight*

| Prozess | Ausführungszeit |
|---------|-----------------|
| A | 30ms |
| B | 50ms |
| C | 20ms |
| D | 30ms |

1.4.2 Round Robin

Das Round Robin(RR) Verfahren ist weitverbreitet. Es wird in abgewandelter Form auch heutzutage in Windows verwendet. Das RR *Scheduling* nutzt Zeitquanten. Diese befinden sich meistens im ein- oder zweistelligen Millisekundenbereich. Jeder Prozess wird nacheinander für die Länge des Zeitquanten ausgeführt und danach dem Prozessor einzogen. Ist der Prozess früher beendet, wird ein neuer Zeitquant eingeleitet. Ist der Prozess nicht beendet, wird er wieder in die Warteschlange am Ende angehängt.

1.5 Threads

In Modernen Computer Systemen gibt es noch Threads neben Prozessen. Threads sind dabei deutlich häufiger vertreten als Prozesse. Auf meinem Computer laufen durchschnittlich 20x mehr Threads als Prozesse. Dies liegt daran, dass Threads als Prozess eines Prozesses gesehen werden. Ein Programm ist nur ein Prozess kann aber viele eigene Threads besitzen. Die Funktionalität von Threads unterscheidet sich nicht viel von Prozessen. Sie werden im *Scheduler* gleich behandelt und es gibt nur geringe Unterschiede bei dem Kontextwechsel.

Threads haben jedoch ein paar Besonderheiten gegenüber Prozessen. Denn im Gegensatz zu Prozessen, welche unabhängig voneinander Laufen, haben Threads diese Beschränkung nicht. Dies liegt daran, dass Threads auf den Stack des Prozessors zugreifen können. Threads teilen sich somit die globalen Variablen des Prozesses, speichern aber eigene lokale Variablen im eigenen Stack. Wegen diesem Prinzip können mehrere Threads mit dem selben Code arbeiten, ohne sich gegenseitig zu behindern. Threads werden als Teil eines Prozesses betrachtet. Deswegen sind sie viel Leichtgewichtiger und können schneller erstellt und beendet werden. Zusätzlich werden Threads in zwei Kategorien unterteilt.

- Nutzer Level Threads. Diese werden mithilfe einer Thread-library implementiert und sind unabhängig von dem Betriebssystem. Folgend können Nutzer Level Threads auch auf einem System ohne unterstütztem Multiprozessing in Betrieb genommen werden. Außerdem ist das Management

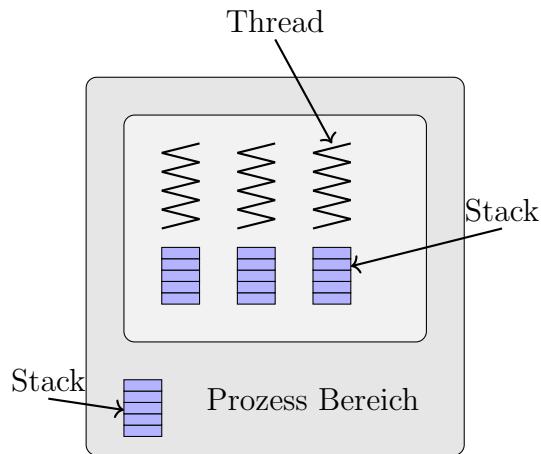


Abbildung 1.4: Threads

von vielen Threads schneller, da keine Systemaufrufe getätigt werden müssen.

- Kernel Level Threads. Diese werden vom Betriebssystem implementiert und auch von diesem verwaltet. Dadurch ergeben sich Vorteile, wie auch Nachteile. Ein Vorteil wäre, dass Prozesse mit einer hohen Thread Anzahl bevorzugt werden könnten. Der Nachteil ist, dass die Verwaltung auf Kernel Ebene deutlich Intensiver ist. Auch muss für jeden Thread ein einzelner Kontrollblock erstellt werden. [3]

1.6 Threads in Java

In Java sind die Threads auf dem Kernel Level implementiert. In Versionen vor 1.2(1998) gab es die so genannten grünen Threads. Diese waren Nutzer Level Threads von der Java VM. Der Vorteil durch das Modernere Modell, ist dass die gesamte CPU genutzt werden kann. Ein Prozess mit mehreren Threads kann somit alle vorhandenen CPU Kerne benutzen. Im alten Modell wurde dies über Umwege gelöst. So wurden ein paar Kernel Level Threads erstellt, welche dann die grünen Threads zugewiesen bekamen.

1.6.1 Implementation in Java

Threads werden in java mithilfe der *java.lang.Thread* Klasse verwirklicht. Es wird dabei zwischen zwei unterschiedlichen Implementierungen unterschieden.
extends Thread: Bei dieser Methode wird eine Klasse erstellt, welche die

Thread Klasse erweitert. Der Vorteil dieses Vorgehens ist, dass alle Einstellungen innerhalb der neuen Klasse getätigt werden können.

```
public class SimpleThread extends Thread {
```

```

public SimpleThread() {
    super();
    //settings
    setPriority(Thread.NORM_PRIORITY);
}

@Override
public void run() {
    System.out.println("Im Running");
}

public static void main(String[] args) {
    SimpleThread t = new SimpleThread();
    t.start();
}
}

```

Das Vorgehen hat seine Probleme. Denn es wird nicht die *run()* Methode von Thread überschrieben, sondern die von Runnable, welches von Thread implementiert wird.

1.6.2 ThreadPool

1.6.3 Thread sicherheit

2 Quellen

MapBild : SynchronousJFXFileChooser : <https://stackoverflow.com/questions/28920758/javafx-filechooser-in-swing> Answer from Sergei Tachenov

3 Programm

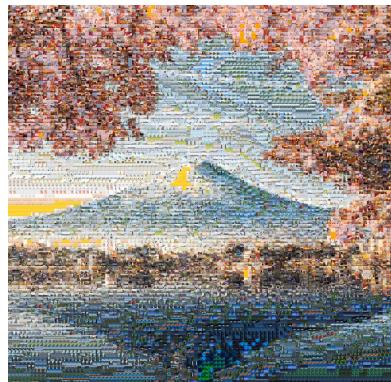
3.1 Einleitung

Um die Methoden der Threads in einer Realen Situation zu nutzen, habe ich mich entschlossen ein Programm, welches stark von Threads profitieren kann, zu programmieren.

Ich habe mich für ein Programm entscheiden, mit dem man ein Bild mit vielen weiteren Bildern rekrieren kann. Es wird demnach ein Mosaik aus Bildern erstellt.



(a) Input



(b) Output

Abbildung 3.1: Funktionsweise des Programmes

Die Anwendung von Threads kommt in dem Programm in vielen Stellen vor. Im folgen werde ich mich auf den Algorythmus der Bild analyse und verarbeitung beziehen. Andere aspekte, wie die implementation des Testmodus und andere Features, die in der App vorhanden sind, werden kurz im Anhang erwähnt.

3.2 Idee

In der genaueren Betrachtung, muss das Programm das eingegebene Bild vereinfachen. Dies wird durch eine Unterteilung des Bildes in Sektoren erreicht. Die Sektoren sind auch die zukünftigen Stellen für die ausgewählten Bilder. Es wird die durchschnittliche Farbe der Sektoren berechnet. Das selbe vorgehen wird auch auf die ausgewählten Bilder angewendet. Dabei ist ein Bild ein Sektor. Dieses vorgehen erlaubt es auch, Datenbanken (im JSON-Format) zu erstellen, da sich die durchschnittliche Farbe der ausgewählten Bilder nicht ändern wird. Anschließend wird mit einem Algorithmus die passenden Bilder für die einzelnen Sektoren berechnet. Dabei unterliegt der Algorithmus der Beschränkung, dass der Nutzer wählen kann, wie häufig ein Bild vorkommen darf. Nach dem berechnen der benötigten Bilder müssen diese für die Verwendung angepasst werden. Es soll schließlich nicht ein 10x10px großer Teil aus einem 4000x3000px Bild verwendet werden. Dazu wird das Bild erst Skaliert und schließlich zurechtgeschnitten. Dabei liegt der Fokus darauf, immer die Mitte des Bildes zu treffen. Das Programm arbeitet die Schritte nachfolgend ab und in dieser Reihenfolge werde ich das Programm nachfolgend erklären.

3.3 Implementation in Java

3.3.1 Berechnen der Sektionen

Das berechnen der Sektionen ist eigentlich ziemlich geradeaus. Die Größe des Bildes ist bekannt und in wie viele Teile dieses aufgeteilt werden soll. Dazu wird z.B. die Breite des Bildes mit der vertikalen Anzahl an Sektoren geteilt. Das Problem liegt darin, dass die Breite oder Höhe des Bildes nicht ein redundant von den gewünschten Sektoren ist. Um das Problem anzugehen wird ersterer Schritt nur mit Integers ausgeführt und der Rest berechnet wird. Übergeblieben sind wie viele Spalte auf der X- und Y-Achse gemacht müssen und der Rest der über bleibt. Im nächsten Schritt werden die Größen der Sektoren in Pixeln berechnet. Dies erfolgt dadurch, dass zwei Arrays für die X- und Y-Achse mit der Anzahl an benötigten Spalten gefüllt werden. Ein weiteres Array mit der Anzahl der Spalten wird aufsteigend nummeriert. Das Array wird von einer Methode gemischt. Eine Abschließende Schleife wird mit der Anzahl des Restes wiederholt und bei jedem Schritt wird ein dann zufälliger Wert aus den Array als Index für das Array mit der Größe der Sektoren genutzt. Das nun zufällige Segment wird um 1 erhöht. Die berechneten Werte werden in einem “splitObj” gespeichert. In diesem werden die Werte in unterschiedlichsten Arten wie z.B. Koordinaten gespeichert.

Diese Koordinaten werden im nächsten Schritt benutzt, um ein zweidimensio-

nales Array an “BufferedImages” aus dem Original Bild zu extrahieren. Dazu wird die “getSubimage()” Methode der Klasse “BufferedImage” verwendet.

3.3.2 Berechnen der durchschnittlichen Farbe

Das berechnen der durchschnittlichen Farbe von Segmenten und Bildern ist die erste Funktion, welche Threads nutzt. Implementationen von mir, welche Threads benutzen, bestehen meistens aus zwei Klassen. Eine Klasse, welche die Threads erstellt und verwaltet und eine weitere, welche den Code zum berechnen der jeweiligen Anforderung hat. In diesem Falle das Berechnen der durchschnittlichen Farbe. Die Klasse “computeAverageColor” besteht aus zwei Methoden. Eine zum Berechnen der Segmente und eine weitere für die gewählten Bilder. Der einzige Unterschied der beiden Methoden ist eine Sicherheitsfunktion in der Methode der Bilder welche wartet, bis genügend RAM verfügbar ist, bevor es das Bild in den RAM lädt. Dies ist wichtig, da mehrere Threads gleichzeitig die Bilder im RAM zum Berechnen halten müssen. Je nach RAM Configuration und Bildergröße kann dies zu Komplikationen führen. Um zuverlässig die Größe des Bildes zu berechnen, kann nicht einfach die Dateigröße verwendet werden. Durch Komprimierungsverfahren wie jpg und png kann die Dateigröße eines Bildes um das 10-fache verkleinern. Um die wirkliche Größe zu berechnen, müssen die Dimensionen mal die Farbtiefe gerechnet werden. Die verwendeten BufferedImages haben eine Farbtiefe von 4byte oder 32bit. Um die Dimensionen eines Bildes auszulesen, ohne das gesamte Bild in den Speicher laden zu müssen, wird die “ImageIO” Klasse von java genutzt. Die “ImageIO” Klasse bietet eine Methode, mit welcher alle “ImageReader” eines Bildes erstellt werden. Ein “ImageReader” benötigt lediglich die Metadaten eines Bildes, welche nur einen Bruchteil der Dateigröße einsprechen, um die Dimensionen des Bildes zu lesen. Jeder der Threads ruft die Methode “getAverage()” der Klasse “calculateAverage” auf. Die Methode verlangt ein “BufferedImage” und ein enum “calculateAverage.Method”, welches unterschiedliche Genauigkeitsstufen zum berechnen beinhaltet. Innerhalb der Methode “getAverage()” wird erst mit Hilfe des enums ein int definiert, welches beschreibt wie die nachfolgenden for-Schleifen erhöht werden. Weiterhin gibt es dre “long”. Diese speichern den gesamten rot, blau und grün wert. Anschließend wird das Bild in der vorherigen definierten Schrittgröße durchgegangen und die rot, grün und blau werte des Pixels gespeichert. Zum Schluss wird ein neues “Color” Object mit den gesammelten werden geteilt durch die Anzahl der Pixel.

3.3.3 Erstellen einer Datenbank

Dieser Schritt wird immer dann durchgeführt, wenn im Programm Bilder ausgewählt wurden, oder eine speicherbare Datenbank erstellt wird. Wenn Datenbanken ausgewählt wurden, wird die eventuell erstellte Datenbank diesen hinzugefügt. Die Klasse “DatabaseObj” benötigt zwei Argumente. Eine Liste an Speicherorten der jeweiligen Bilder und ein Array an berechneten durchschnittlichen Farben. Innerhalb des “DatabaseObj” werden die beiden Werte der Listen in “fileAndColor” Objekten gespeichert. Wie der Name impliziert beinhalten die Objekte den Pfad des Bildes und den dazugehörigen Farbwert. Durch dieses Vorgehen kann ein BinarySort Algorithmus aus das neue Array angewendet werden. dazu hat die “fileAndColor” Klasse das “Comparable” interface implementiert. Für spätere Verwendung gibt es in den Objekten auch noch einen Zähler, welcher speichert, wie häufig das Bild in der Berechnung verwendet wurde. Alle Variablen der Klasse “fileAndColor” sind durch getter-setter-Methoden aufzurufen. Diese habe jedoch die Besonderheit, dass sie das “synchronized” Schlüsselwort besitzen (Hyperlink zu Kapitel Threads).

Besonders werde ich auf die Methode “compareTo()”, des Interfaces Comparable eingehen. In der Methode werden zwei Farben verglichen. Welche größer und kleiner ist. Die Schwierigkeit liegt darin ein Ergebnis zu bestimmen, wenn vier Werte verglichen werden müssen. Der alpha-Wert, rot-Wert, grün-Wert und blau-Wert. Um ein einfachen und zuverlässigen Vergleich zu machen, nutze ich die Methode “getRGB()” der “Color” Klasse. Interessant ist jedoch, was diese Methode macht. In der Klasse “Color” werden alle vier Werte in einem einzigen Integer gespeichert. Das liegt an der Genauigkeit, womit Farben gemischt werden. Jeder Wert befindet sich in einem Bereich von 0 bis 255. Dies entspricht genau einem Byte. Ein Integer wiederum besteht aus vier Bytes. Die vier Farbwerte werden sozusagen in die vier Bytes des Integers eingesetzt. In der Reihenfolge “ARGB”.

| | | | | | |
|------------|----------|----------|----------|---|----------|
| alpha(255) | rot(224) | grün(64) | blau(16) | ■ | -2080752 |
| 11111111 | 11100000 | 01000000 | 00010000 | | |

Durch dieses Vorgehen wird aus vier Werten ein zuverlässiger Wert erstellt, welcher zum sortieren geeignet ist.

3.3.4 Berechnen der besten Bilder

Das Berechnen der besten Bilder für die jeweiligen Sektoren wird in der Klasse “compareColor” mit der Methode “compare()” ausgeführt. Die Methode gibt ein zweidimensionales “File”-Array zurück. Dies steht dabei für die einzelnen Sektoren. Innerhalb der Methode wird für jeden Sektor ein “Runnable” erstellt. (siehe Kapitel Threads). Da es ein Limit für die Verwendung der Bilder gibt, sollen diese gleichmäßig verteilt werden. Um einen solchen Effekt zu erhalten wird eine Liste an “Runnables” erstellt und gemischt.

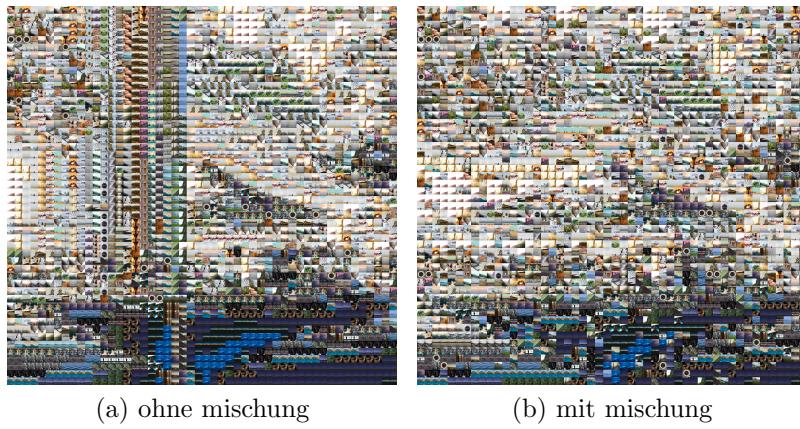


Abbildung 3.2: Unterschied zwischen mischen und nicht mischen

Jeder “Runnable” sucht demnach nach dem besten Bild, welches noch verfügbar ist. Es werden dazu alle vorhandenen Datenbanken durchgegangen und für jede einzelne das beste Bild gesucht. Die besten aus allen Datenbanken werden dann verglichen und der beste wird gewählt. Um die beste Farbe einer Datenbank zu berechnen kann die java Methode “Arrays.binarySearch” verwendet werden. Diese Methode findet nicht nur den gleichen Wert, sondern auch einen index, an dem es den gesuchten Wert einsetzen würde. Anhand von dem Index wird dann im Array jeweils das nächste beste Bild, welches noch nicht zu häufig verwendet wurde, gesucht. Die beiden gefundenen Werte, werden auf Nähe zum gesuchten Wert überprüft. Der nähere wird dementsprechend auserwählt.

3.3.5 Skalieren der Bilder

Das Skalieren der Bilder ist in drei Schritte einzuteilen. Das Management der Bilder, Berechnen der Zielgröße und das Skalieren an sich. Ich werde auf jeden Bereich individuell eingehen.

Management

Da Ein Bild mehrfach ausgewählt werden kann, ist es wichtig ein Management System zu haben, welches verhindert, dass das selbe Bild nicht häufiger als nötig skaliert wird. Das Skalieren ist der Zeitaufwendigste Prozess, daher sollte dieser minimiert werden. Jedes Bild kann in vier unterschiedlichen Größen verwendet werden. Die Größen kommen aus dem in (siehe Berechnen der Sektionen) beschriebenen vorgehen. Werden die Bilder nur in einer einheitlichen Größe skaliert Entstehen schwarze Linien im Bild.



Abbildung 3.3: vierfache Skalierung der Bilder

Beim Managen wird demnach überprüft, ob ein bestimmtes Bild schon in der jeweiligen Größe vorhanden ist. Dazu wird eine Instanz der Klasse “ScaledImages” erzeugt. Diese Instanz fungiert als gemeinsamer Speicherort aller skalierten Bilder. Die Klasse hat ein zweidimensionales Array des Types “ImageWithName” und eine Methode “exists()”. Die Methode “exists()” sucht mithilfe von dem Pfad des Bildes und den gewünschten Dimensionen in dem Array nach bereits skalierten Bildern. In dem Fall, dass etwas gefunden wird, werden die Koordinaten des gefunden Bildes zurückgegeben. Der Skalier-Thread speichert dan eine Referenz auf das gefundene Bild ab.

Ein Problem beinhaltet das System jedoch noch. Es könnte zu der Situation kommen, dass mehrere Threads gleichzeitig mit dem gleichen Bildauftrag gestartet werden. Anfangs überprüft jeder Thread, ob sie arbeiten dürfen, oder das gewünschte Bild noch nicht existiert. Alle Threads werden davon ausgehen, dass die Arbeiten dürfen, da das Skalieren des Bildes und Speichern

dessen vergleichsweise länger dauert als einen neuen Thread zu starten. Doch dadurch werden nicht nur Ressourcen unnötig verbraucht, sondern auch Fehler in der Bilddatei können entstehen, da mehrere Threads gleichzeitig eine Datei auslesen.

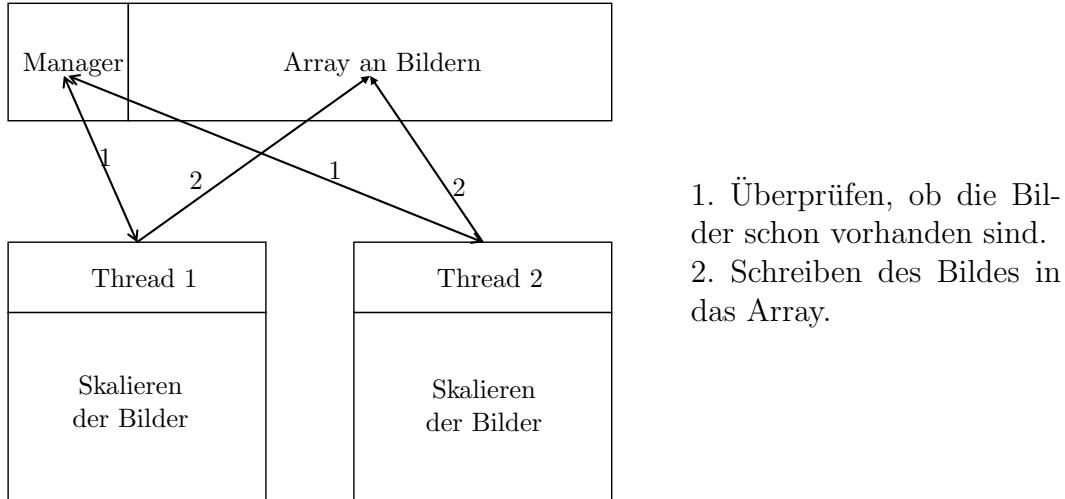


Abbildung 3.4: Threads mit Manager

Um ein solches Verhalten zu verhindern, versucht ein Thread ein Bild zu reservieren, wenn es laut Manager frei ist. Ein Reserviertes Bild, wird als bereits existierendes gewertet. Das Referenz System funktioniert weiterhin, da keine neuen Objekte erstellt werden, sondern diese nur mit dem Bild befüllt werden. Die Array Struktur wird durch rasantes steigen des Speicherverbrauchs einzelner Objekte nicht beeinflusst. Das liegt unter anderem an dem “GarbageCollector” von Java, welcher auch in Stande ist Arrays aus ihrem sonst linearem Besetzen eines Speicherblockes im Arbeitsspeicher aufzuteilen, um unter anderem größer werdenden Objekten Platz zu schaffen. Dabei werden interne Referenzen des Arrays überschrieben, um zu dem verschobenen Objekt zu zeigen.

Berechnen der Zielgröße

Das Ziel ist, die Bilder so gut wie möglich zu verwerten. Dazu müssen unterschiedliche Kriterien erfüllt werden. Erstens sollte das Bild nicht verzogen werden, zweitens sollte immer der größtmögliche Bereich des Bildes verwendet werden.

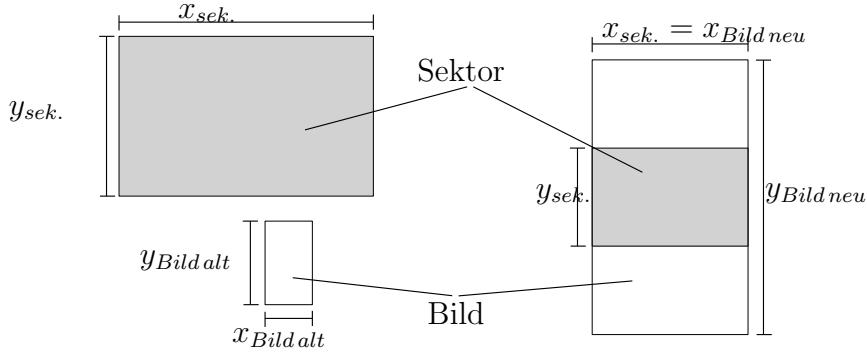


Abbildung 3.5: Berechnen Zielgröße

Um das zu erreichen wird das Bild erst auf die Größe des gewünschten Sektors skaliert. Anschließend werden die Ränder abgeschnitten. Beim Skalieren wird versucht eine x-Achse des Bildes der y-Achse des Sektors anzupassen. Damit kann sichergestellt werden, dass das Bild seine Verhältnis beibehält. Wie in der oberen Grafik zu erkennen ist, wird ein Vertikal ausgerichtetes Bild in einen Horizontalen Sektor eingesetzt. Dazu wurde die x-Achse des Bildes auf die des Sektors skaliert. Die Formel zum Berechnen der y-Achse ist wie folgt.

$$y_{Bild\ neu} = y_{Bild\ alt} \cdot \frac{x_{sek.}}{x_{Bild\ alt}} \quad (3.1)$$

Wenn die neu berechnete y-Achse kleiner ist als die des Sektors wird das selbe mit der x-Achse gemacht. Im letzten Schritt wird ein Abschnitt mittig aus dem skaliertem Bild mit den Abmaßen des Sektors geschnitten.

Skalieren des Bildes

Einleitend muss ich zu diesem Abschnitt sagen, dass ich die Folgenden Methoden nicht selber implementiert habe. Ich nutzte dazu die imgskalr library von Riyad Kalla, welche auf der Bilinearen und Bikubischen Interpolation von Javas “Graphics2D” beruht. Im folgenden werde ich die Bilineare Interpolation erklären. Auf die Bikubische werde ich nur kurz eingehen.

Bei dem Hochskalieren von Bildern stößt man auf das Problem, dass Farbwerte erfunden werden müssen. Andere Algorithmen erfinden keine neuen Farben und sind gut für Pixel Bilder geeignet. Für Landschaftsbilder würde dies lediglich ein verpixeltes Bild ergeben. Um neue Farbwerte zu Berechnen, wird bei der Bilinearen Interpolation linear Farbwerte berechnet.

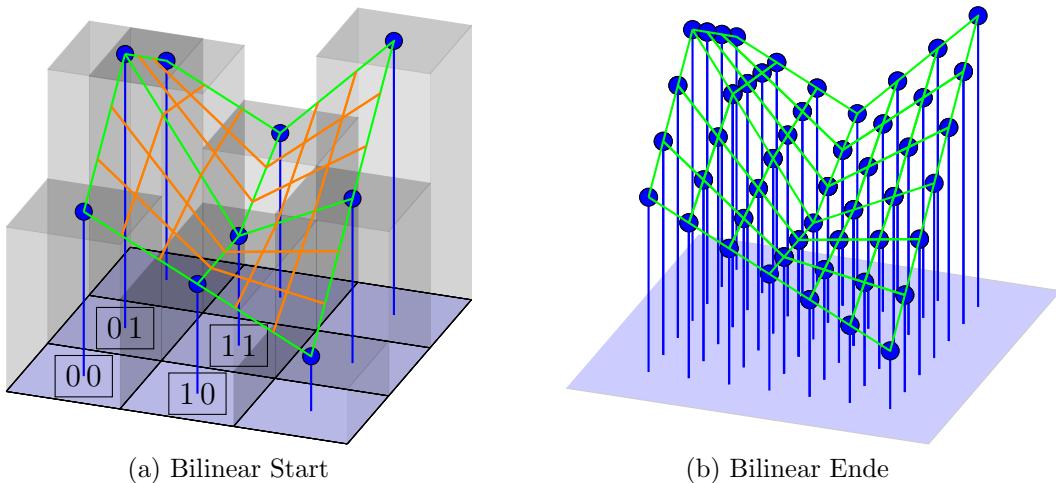


Abbildung 3.6: Bilineare Interpolation

Die unterschiedlichen Höhen in der der Grafik repräsentieren die Unterschiedlichen Farbwerte der Pixel. Das 3x3 Bild wird um den Faktor 3 skaliert. Zuerst wird ein Netz zwischen den einzelnen Werten gespannt. Die neuen Farbwerte entstehen an den Schnittpunkten des Netzes. Der Farbwert lässt sich somit mit

$$W_{neuX} = (W_{10} - W_{00}) \cdot \frac{n_x}{S_{fak.}} \quad (3.2)$$

$$W_{neuY} = (W_{11} - W_{01}) \cdot \frac{n_y}{S_{fak.}} \quad (3.3)$$

$$W_{neu} = W_{00} + (W_{neuX}) + (W_{neuY} - W_{neuX}) \cdot \frac{n_y}{S_{fak.}} \quad (3.4)$$

In den Gleichungen 3.2 und 3.3 werden die Start- und End-Punkte einer Netzelinie berechnet. n_x und n_y sind Zähler, welche die gewünschte Position in dem Teilnetz angibt. Das Maximum der beiden ist $S_{fak.} + 1$. $S_{fak.}$ ist dabei der Faktor, um wie viel das Bild skaliert werden soll. Der Unterschied zwischen Bilinearer Interpolation und Bikubischen ist, dass bei der Bilinearen lediglich zwei

Linien aus jeweils zwei Punkte benutzt werden. Bei der Bikubischen werden vier kubische Spline Funktion aus jeweils vier Punkten benutzt. Der Vorteil der Bikubischen ist, dass das resultierende Bild einen höheren Kontrast bekommt. Auch können Komplexere Formen besser berücksichtigt werden.

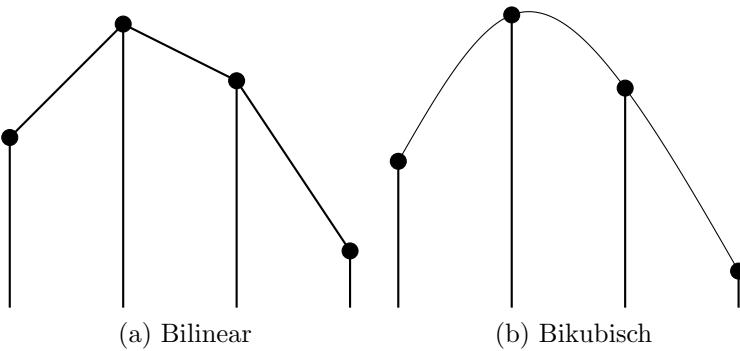


Abbildung 3.7: Vergleich Bilinear und Bikubisch

Um die beste Qualität bei dem Runterskalieren zu erhalten, wird das Bild häufiger skaliert. Dies verwaltet die imgskalr library. Das Bild wird wiederholt um $\frac{1}{7}$ seiner Breite und Höhe verkleinert. Der Wert wurde nicht Mathematisch berechnet, sondern wurde durch testen herausgefunden.

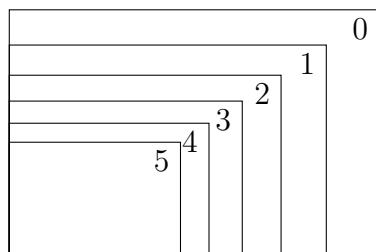


Abbildung 3.8: Inkrementelles Skalieren

4 Laufzeitanalyse

4.1 Einleitung

Um die Effizienz des Programms beurteilen zu können, werde ich die einzelnen Algorithmen betrachten. Dazu werde ich das erstellen einer Datenbank, berechnen der durchschnittlichen Farbe, skalieren der Bilder und das berechnen der besten Bilder vergleichen. Der Test kann mit einer UI im Programm durchgeführt werden. Es wird ein Test mehrfach durchgeführt und bei jedem neuem Durchgang die Anzahl an Bildern erhöht. Das Testen kann mit generierten Bildern und mit zufälligen Bildern der Internetseite “<https://picsum.photos>” durchgeführt werden. Alle folgenden Tests werden mit zufälligen Bildern durchgeführt, da diese die echte Benutzung am besten simulieren. Bei den Generierten Bildern sinken die Berechnungszeiten stark, da weniger Farbkomplexität gegeben ist. Bei jeder Simulation wird auch der benötigte Arbeitsspeicher gespeichert, da das Programm sehr Arbeitsspeicher intensiv werden kann. Die Laufzeit des Berechnen der durchschnittlichen Farbe, das erstellen einer Datenbank und das berechnen der besten Bilder ist Linear. Folglich ergibt sich die Laufzeitkomplexität von $s(n)$. Wobei n die Anzahl der verwendeten Bilder ist. Die Größe der Bilder wurde nicht beachtet, da diese allein durch die zweidimensionale Natur von Bildern eine Quadratische Laufzeit ergeben würden.

$$s(n) \in \mathcal{O}(n) \quad (4.1)$$

Die Laufzeit des Skalieren ist abhängig von dem genutzten Algorithmus.

Alle Tests wurden mit dem folgenden Computer durchgeführt.

| | |
|-----------------|---|
| CPU | AMD Ryzen 7 2700 8Core 16Threads 3,65GH |
| Festplatte | <u>Corsair MP600 2T NVMe M.2 SSD</u> Lesen: 3,4GB Schreiben: 3,2GB |
| Arbeitsspeicher | <u>GSkill 2x16GB</u> Takrate: 3,2GH |
| JavaVM | <u>java 17 Runtime Environment</u> Arg: -Xmx25G |

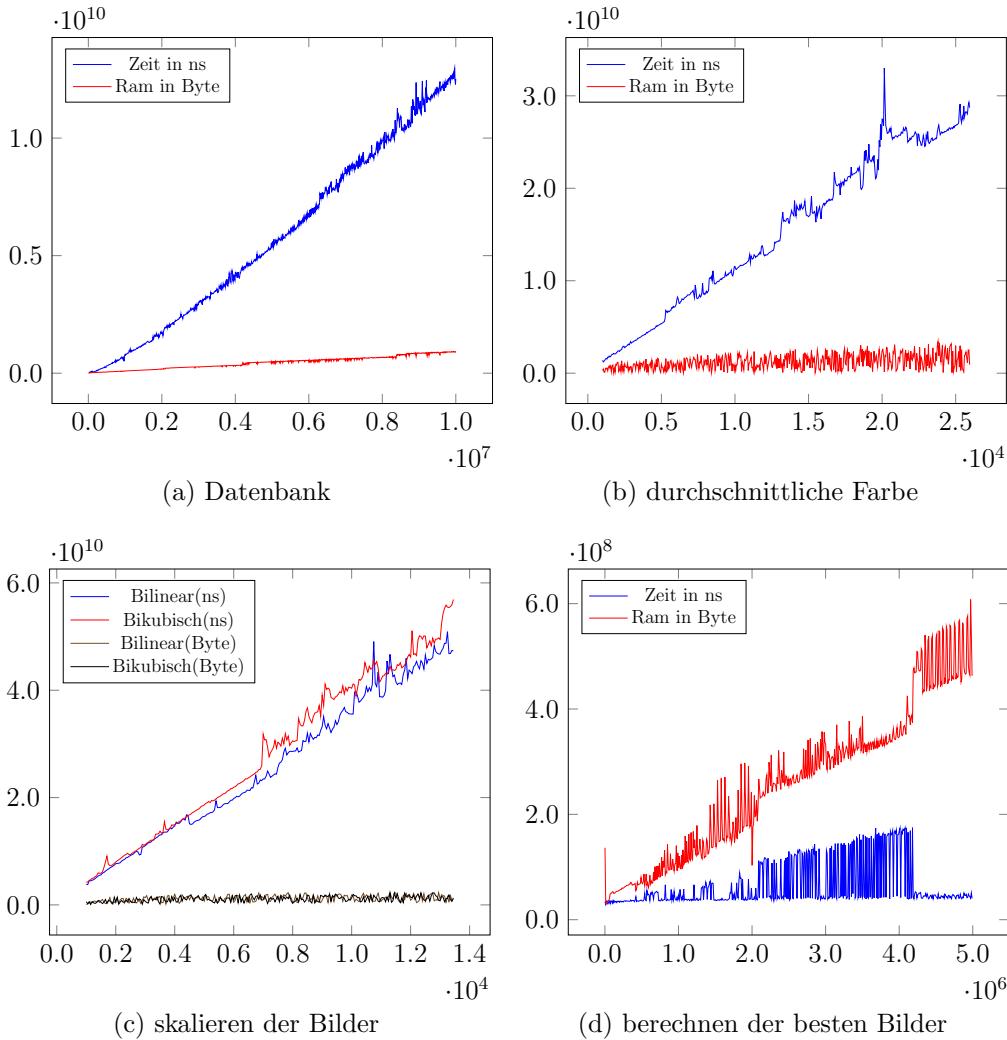


Abbildung 4.1: Vergleich Laufzeiten und Arbeitsspeicher

4.2 Testungen

Test 4.1a wurde von 10000 bis 10000000 Bilder durchgeführt. Die Zeit erhöht sich um 1250ns pro Bild. Die Arbeitsspeichernutzung erhöht sich um 87Byte pro Bild. Die Arbeitsspeichernutzung ist dabei auch stark von der Länge des Speicherortes des Bildes abhängig.

Test 4.1b wurde von 1000 bis 25000 Bilder durchgeführt. Die Bilder hatten eine Größe von 500x500px. Die Laufzeit für ein Bild liegt bei 1,09ms. Für jedes weitere Bild werden 88kByte verbraucht. Beide Werte verhalten sich linear zu der Anzahl der Bilder.

Das Skalieren der Bilder in 4.1c wurde jeweils mit einem Bilinearen Algorithmus und einem Bikubischen Algorithmus durchgeführt. Dazu wurden 500x500px große Bilder auf 100x70px skaliert. Der Test wurde mit 1000 Bildern gestartet und mit 12500 beendet. Es ist zu erkennen, dass nur ein geringer Laufzeitunterschied von $\Delta t = 4,2ms - 3,7ms = 0,5ms$ zwischen den beiden Algorithmen besteht. Dieser geringe Unterschied kommt daher, dass beide Algorithmen sich

sehr ähnlich verhalten. Der Unterschied liegt lediglich bei der komplexeren Berechnung in dem Bikubischen Algorithmus, welche nur einen Teil der Laufzeit beansprucht. Der Arbeitsspeicherverbrauch ist bei beiden Algorithmen gleich und erreicht 136KB.

Der schnellste Prozess ist das berechnen der Bilder. Dies liegt an der zuvor erstellten Datenbank und ihrem sortierten Aufbau. Dabei wird eine Laufzeit von 2,31ns pro Bild erreicht. Der Test wurde mit 5000000 Bildern durchgeführt. Die Arbeitsspeichernutzung ist die selbe wie in Test 4.1a, da das berechnen der Bilder auch Datenbanken braucht.

Als nächstes werde ich alle Algorithmen gemeinsam Testen. Dazu werde ich als Referenzbild ein 4890x3263px Bild¹ verwenden. Ich werde die Datenbank nur einmal erstellen, da die Laufzeit von dieser sich nicht verändert wird. Bei jedem Test, wird das zu berechnete Bild um 3 skaliert. Es sollte demnach ein 14670x9789px Bild entstehen.

| Test Typ | Spezifikationen | Zeit |
|----------------|--|---|
| Datenbank | Bilder: 500000 Größe: 200x300 | 10 min 26 sek |
| | Multiplikator: 3 ^a Max.: 50 ^b Qualität: Ultra ^c | |
| Bild erstellen | Aufteilungen: 500x500 Aufteilungen: 700x700 Aufteilungen: 999x999 | 02 min 20 sek 04 min 03 sek 21 min 38 sek |

^a Das resultierende Bild wird drei mal größer sein.

^b Die maximale Anzahl, die ein Bild verwendet werden darf.

^c Die beste Einstellung. Es werden bei der Analyse keine Pixel übersprungen und Bikubische Skalierung wird verwendet.

Es ist zu erkennen, dass die Steigung von Test 1 auf Test 2 den Erwartungen entspricht. Die Sektoren Anzahl hat sich verdoppelt, dementsprechend auch die Laufzeit. Bei dem Sprung von Test 2 auf Test 3 ist eine Vervierfachung anstelle einer erwarteten Verdopplung aufgetreten. Eine Erklärung für das Verhalten könnte die ansteigende Datenmenge, welche das Optimum des Test Computers überschreitet. Eine Version mit 999x999 Sektoren und einem Multiplikator von 6 kann unter folgendem Link jeweils mit 50% Transparenz und ohne angesehen werden. Ich empfehle einen Computer mit 16GB Arbeitsspeicher, um sich die Bilder anzusehen, da sie 1GB als PNG verbrauchen.

<https://shorturl.at/imHV3>

¹ Bild von <https://media.jaguar.com/de-de/news/2020/11/jaguar-f-type-r-cabrio-siegt-bei-den-sport-auto-awards-2020>

Literatur

- [1] BRAUN, C.: *Betriebssysteme kompakt*. Springer Berlin Heidelberg, 2020.
- [2] CHIDAMBARAM, P. V.: *Linux Kernel Scheduler*. https://www.youtube.com/watch?v=Jl_0W4o0xao, 2020.
- [3] DUSEY, A.: *Threads and its types in Operating System*. <https://www.geeksforgeeks.org/threads-and-its-types-in-operating-system/>, 2021.
- [4] JAMES, S.: *Understanding Linux Scheduling*. <https://www.linkedin.com/pulse/20140629145049-21586023-understanding-linux-scheduling>, 2014.
- [5] KERRISK, M.: *sched - overview of CPU scheduling*. <https://man7.org/linux/man-pages/man7/sched.7.html>, 2021.
- [6] LOVE, R.: *What is a tickless kernel?*. <https://www.quora.com/What-is-a-tickless-kernel>, 2013.
- [7] MANDL, P. D. P.: *Prozesse und Threads*. https://www.wirtschaftsinformatik-muenchen.de/wp-content/uploads/Peter%20Mandl/Lehrveranstaltungen/SoSe%2015/Wirtschaftsinformatik/05_Prozesse_u_nd_Threads.pdf, 2015.
- [8] REBEIRO, P. C.: *Lecture 22 Completely Fair Scheduling*. <https://www.youtube.com/watch?v=scfD0of9pww>, 2017.
- [9] TORVALDS, L.: *Linux Documentation*. <https://github.com/torvalds/linux/tree/master/Documentation/scheduler>, 2022.
- [10] TORVALDS, L.: *Linux Kernel*. <https://github.com/torvalds/linux/tree/master/kernel/sched>, 2022.
- [11] WILLIAMS, L.: *CPU Scheduling Algorithms in Operating Systems*. [https://www.guru99.com/cpu-scheduling-algorithms.html#:~:text=Six%20types%20of%20process%20scheduling%20algorithms%20are%3A%20First%20Come%20First,%2C%206\)%20Multilevel%20Queue%20Scheduling.](https://www.guru99.com/cpu-scheduling-algorithms.html#:~:text=Six%20types%20of%20process%20scheduling%20algorithms%20are%3A%20First%20Come%20First,%2C%206)%20Multilevel%20Queue%20Scheduling.), 2022.