



Evangelisches Gymnasium Werther

Besondere Lernleistung

Prozesse und Threads

anhand einer Rasterbildsoftware

Verfasser: **Philipp Bleimund**

Kurs: Grundkurs Informatik

Betreuende Lehrkraft: Herr Christian Möllenbrock

Abgabetermin: 07.04.2022

Inhaltsverzeichnis

1 Einleitung	3
2 Threads und Prozesse	4
2.1 Aufbau von Prozessen	4
2.2 Verwalten der Prozesse	6
2.3 Funktionsweise des Schedulers	9
2.3.1 Completely Fair Scheduling	11
2.3.2 Round Robin	13
2.4 Threads	14
2.5 Threads in Java	15
2.5.1 Implementation in Java	15
2.5.2 Thread-Pool	16
2.5.3 Threadsicherheit	17
3 Programm	19
3.1 Idee	20
3.2 Implementation in Java	20
3.2.1 Berechnen der Sektoren	20
3.2.2 Berechnen der durchschnittlichen Farbe	21
3.2.3 Erstellen einer Datenbank	22
3.2.4 Berechnen der besten Bilder	23
3.2.5 Skalieren der Bilder	24
4 Laufzeitanalyse	29
4.1 Testungen	30
5 Schluss	32
5.1 Aussicht	32

1 Einleitung

Eines meiner größten Hobbys ist das Programmieren. Ich kam nicht herum, in Kontakt mit Threads und Prozessen zu geraten. Besonders bei dem Projekt, meines Bilder Mosaiks. Über dessen Implementation wird sich auch ein Teil meiner Arbeit handeln. Als ich eine Dokumentation über Linus Torvalds gesehen habe ist mir ein Zitat von ihm nicht mehr aus dem Kopf gegangen.

*The Linux philosophy is ‘Laugh in the face of danger’. Oops. Wrong One.
‘Do it yourself’. - Linus Torvalds.*

Anstelle darauf zu warten, dass das Wissen auf mich zufliegt, werde ich es in die Hand nehmen und es mir im Zuge dieser Arbeit aneignen. Dieses wird in dem ersten Teil der Arbeit behandelt. Das Programm wird dazu in Java verfasst und für die grafische Oberfläche werde ich Java-swing verwenden. Das Projekt werde ich mit Maven¹ bauen und verwalten.

¹ Maven ist ein in Java geschriebenes Werkzeug, mit welchem sich Java Projekte und deren Bibliotheken verwalten lassen

2 Threads und Prozesse

Prozesse sind die Ausführung eines Programmes auf dem Prozessor. Jedoch kann ein Prozessor maximal einen 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 realem Kern zugeordnet werden. Jedoch wird nicht jeder Prozess gleich lange einem Kern zugeordnet. Die Prozesse konkurrieren um ihre Zeit. Mein Computerspiel soll nicht die gleiche Zeit bekommen, wie meine Stoppuhr-App. Dieser Wechsel zwischen den einzelnen Prozessen nennt man auch Kontextwechsel. Der Kontext des Kerns ändert sich demnach.

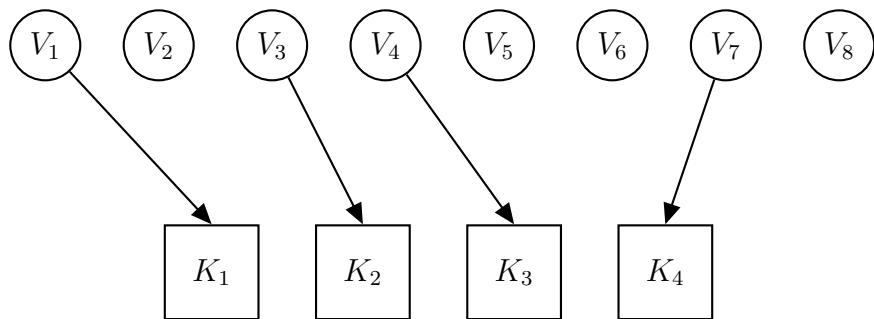


Abbildung 2.1: Aufteilung von Virtuellen Threads auf Kernel

2.1 Aufbau von Prozessen

Prozesse müssen noch ein wenig mehr als nur ein Stück Quelltext 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. Er besteht aus folgendem:

- Das auszuführende Programm
- Die Daten des Programms: umfasst unter anderen die globalen Variablen
- Einem 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: verwaltet den Zugriff auf den Arbeitsspeicher

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

- Instruction Pointer - die Speicheradresse des nächsten Befehls
- Instruction Register - der aktuelle Befehl
- Stackpointer - speichert das Ende des Stacks
- Basepointer - Speicheradresse des aktuellen Elements im Stack
- Akkumulator - speichert Ergebnisse der ALU

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

- Ort in der Prozesstabelle
- PID - Prozessnummer
- Prozesszustand
- Priorität
- Eltern- oder Kindprozesse
- Zugriffsrechte - Linux: -20 bis 19
- Erlaubte Ressourcenmengen - bsp. Maximaler RAM Verbrauch
- Verwendete Dateien - um zu verhindern, dass mehrere Prozesse an einer Datei arbeiten
- Zugeordnete Geräte - Maus, Tastatur, ...

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

2.2 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, siehe Abbildung 2.2. Auch hat ein Prozess deutlich mehr Zustände in einem modernen Betriebssystem als nur *untätig* und *rechnend*. Dazu wird heutzutage meistens das Prozessmodell wie in Abbildung 2.3 oder eine modifizierte Variante verwendet. Linux als Beispiel verwendet ein *8-Zustands Prozessmodell*, welches das Modell mit einem *kernel rechnend* Zustand erweitert.

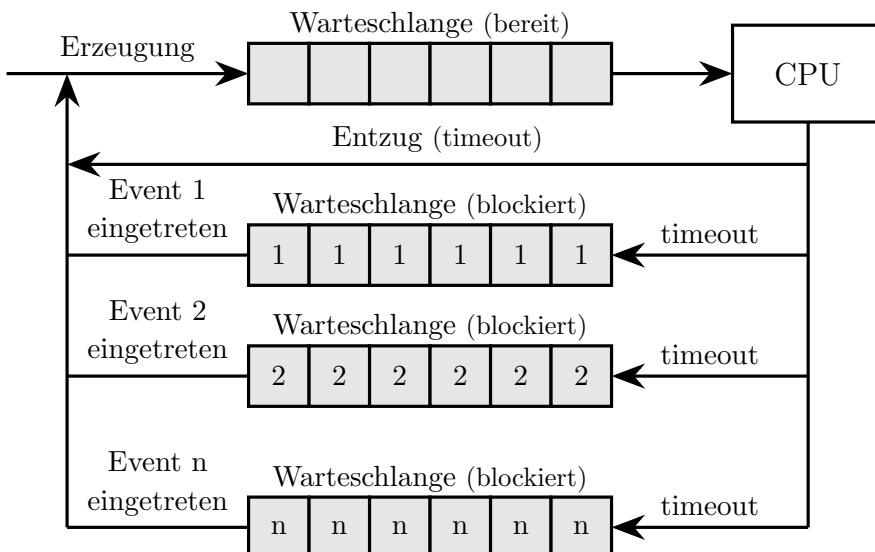


Abbildung 2.2: Warteschlangensystem¹

¹ Grafik in Anlehnung an [1, Seite 154]

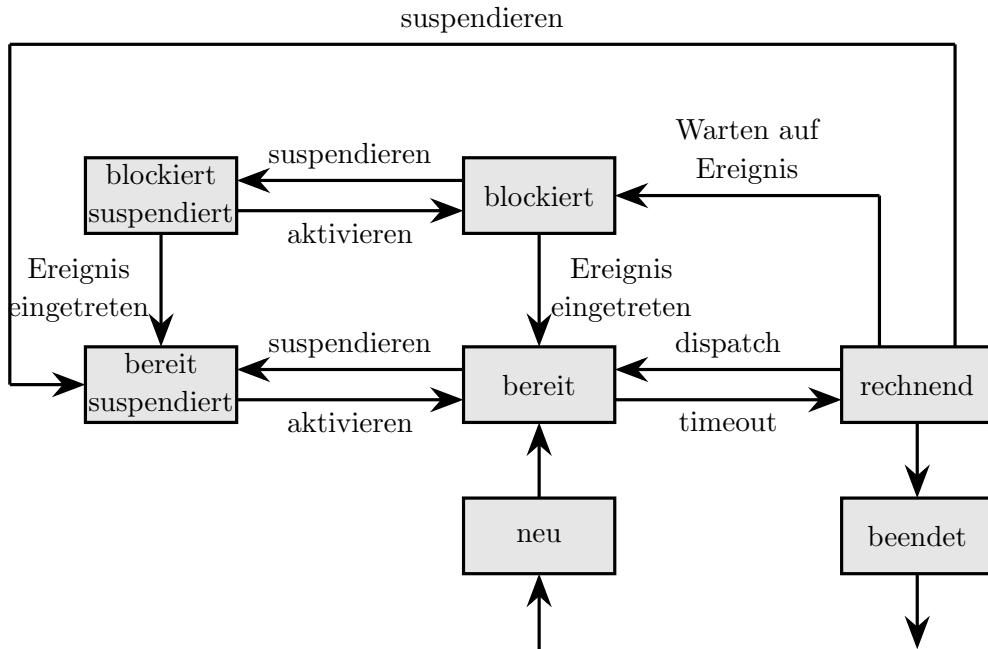


Abbildung 2.3: 7-Zustands Prozessmodell¹

Wie in der Einleitung schon angesprochen sind die Zustände *bereit* und *rechnend* die wichtigsten Zustände. Mit diesen alleine könnte ein Betriebssystem funktionieren. In diesem System gibt es nur eine Warteschlange, in der sich alle Prozesse des Zustandes *bereit* befinden. Idealer Weise implementiert der *Scheduler*² einen Algorithmus, welcher die Priorität der Prozesse berücksichtigt. Wie schon erwähnt muss sich der *Dispatcher*³ noch um weitere Zustände kümmern. Diese und ihre Beziehungen sind in Grafik 2.3 zu finden. Zwei davon wären *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 entscheidender 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 bereitzustellen, gibt es den Zustand *blockiert*. In diesen wechselt ein Prozess nach dem berechnen und kann aus diesen sich wieder in die Warteschlange der *bereiten*-Prozesse einordnen. In Grafik 2.2 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

¹ Grafik in Anlehnung an [1, Seite 156]

² Programm zum Managen der Warteschlangen.

³ Programm zum Ausführen der Prozesswechsel.

blockiert suspendiert und *bereit suspendiert* eine Möglichkeit geschaffen, selten genutzte Prozesse aus dem Arbeitsspeicher in den Massenspeicher¹ zu verschieben. Wie die Namen schon implizieren werden Prozesse in den Zuständen *blockiert* und *bereit* jeweils suspendiert und aktiviert. 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 *Warteschlangensystem*² für die suspendierten Prozesse. Dieses beinhaltet keinen ?Zugriff auf die CPU, sondern kann die Prozesse maximal aktivieren und in den Arbeitsspeicher verschieben. [1]

¹ Spezielle Partitionen auf einer Festplatte. Auch *swap* genannt.

2.3 Funktionsweise des Schedulers

Der *Scheduler* ist ein sehr wichtiges und mächtiges Stück Quelltext. Es regelt 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 zwischen ihm und der CPU. Jedes Programm muss über den Kernel um arbeiten 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 extern getriggert. Der *Scheduler* wird entweder von einem beendeten Prozess oder nach einer Zeitunterbrechung getriggert. 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.[8]

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 hat ein Prozess, bis zu seiner Fertigstellung, volle Kontrolle über die CPU. Der *Scheduler* führt erst den Kontextwechsel nach dessen Vollendung aus. Es kann Situationen geben, bei denen ein Prozess sich nicht selber 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. Es lassen sich nicht immer alle Kriterien miteinander vereinbaren und es ist dem Entwickler überlassen, welche er bevorzugt. Die Kriterien wären wie folgt: [19]

- 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 Hertz Anzahl einer CPU

2.3.1 Completely Fair Scheduling

*Completely Fair Scheduling*¹ ist, wie der Name schon impliziert, eine Form des *Ideal Fair Scheduling*. Es besagt, dass versucht wird die Prozesse gleich lange Arbeiten zu lassen. In den Tabellen 2.1 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. [15, Ab 1:35]

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 2.1: *Fair Scheduling (FS)*²

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. Prozesse wie Tracker³ schadet der Kontextwechsel nicht allzu sehr. Der *Scheduler* addiert auf die alte *vruntime* die Zeit Δt , die der Prozessor 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 mit einer Laufzeit von $\mathcal{O}(1)$ den Prozess mit der kleinsten *vruntime* zu finden. Durch das Wählen der kleinsten *vruntime* wird das Prinzip des *Fair Schedulings* eingehalten. [15, Ab 5:22]

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

² Grafik in Anlehnung an [15, Minute 1:35]

³ Programm um etwas aufzuzeichnen (Bspw. Anzahl der Tastenanschläge)

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*. [17]

- **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 *nice* Werten von 11 und 12, wäre 55% und 45%. Bei Prozessen mit -4 und -5 bleibt die Verteilung gleich. Die neue *vruntime* wird mit Formel 2.1[18] berechnet.

$$\text{vruntime} += \text{delta_exec} \cdot \frac{\text{weight}}{\text{lw.weight}} \quad (2.1)$$

Dabei ist *weight* mit 1024 oder *nice_0* definiert. *lw.weight* ist die Priorität als *weight*. Das kann aus dem Array in dem Listing 2.1[18] 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 2.1: Linuxkernel core.c(v5.17-rc3) *nice* Werte als *weight*

2.3.2 Round Robin

Das Round Robin(RR) Verfahren ist weitverbreitet. Es wird in abgewandelter Form auch heutzutage in Windows verwendet.[7] 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 entzogen. 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, Seite 176f.]

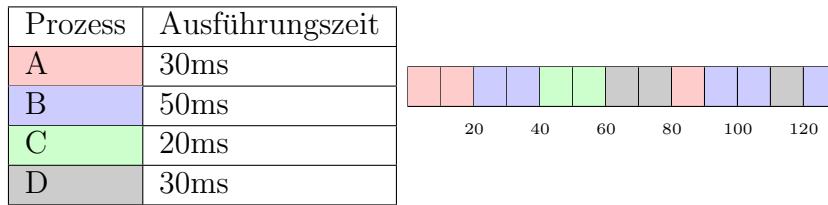


Abbildung 2.4: Round Robin Beispiel

In 2.4 werden vier Prozesse in der vorgegebenen Reihenfolge abgearbeitet. Die Zeitquanten in dem Beispiel sind 20ms lang. Prozess C benötigt nur ein Zeitquanten und wird beendet. Prozesse A und D benötigen 1.5 Zeitquanten. Sie arbeiten beim zweiten Mal nur 10ms und der nächste Prozess kann nachrücken. Prozess B benötigt am meisten Zeit und wird dreimal ausgeführt. Erst beim dritten Mal kann er beendet werden.

Das Round Robin Verfahren kann noch mit Prioritäten erweitert werden. Dieses werde ich am Beispiel des *Schedulers* von Windows erklären. Windows verwendet ein prioritätengesteuertes *Scheduling*¹. Das RR Verfahren wird dabei immer auf die selbe Priorität angewendet. Die höhere Priorität wird immer bevorzugt. Dazu wird für jede Priorität eine eigene Warteschlange angelegt. Um das Verhungern eines Prozesses zu verhindern, nutzt Windows noch variable Prioritäten. Bei den variablen Prioritäten hat Windows noch die Möglichkeit selber zu ermitteln welche Prozesse wichtiger sind.[7, Folie 9-12]

Mit dem Round Robin Verfahren kann grundsätzlich eine faire Umgebung geschaffen werden. Werden jedoch Prioritäten ohne Variation bevorzugt, kann es zu einem unfairen System werden. Auch sind IO-lastige² Prozesse benachteiligt. Sie benötigen selber nur wenig Ausführungszeit, müssten aber für das reibungslose Funktionieren häufiger Zugriff auf den Prozessor erhalten. Im Round Robin *Scheduling* müssen sie warten, bis rechenintensive Prozesse den Zeitquanten vollständig verbrauchen, bevor sie wieder an der Reihe sind.

¹ Die Prioritäten von Windows reichen dabei von 0 bis 31

² IO = InputOutput. Ein Prozess wartet z.B. auf eine Eingabe des Benutzers.

2.4 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. Es liegt daran, dass Threads auf den Stack des Prozesses zugreifen können. Threads teilen sich somit die globalen Variablen des Prozesses, speichern aber eigene lokale Variablen im eigenen Stack. Aufgrund dieses Prinzip können mehrere Threads mit dem selben Quelltext arbeiten, ohne sich gegenseitig zu behindern.[20]

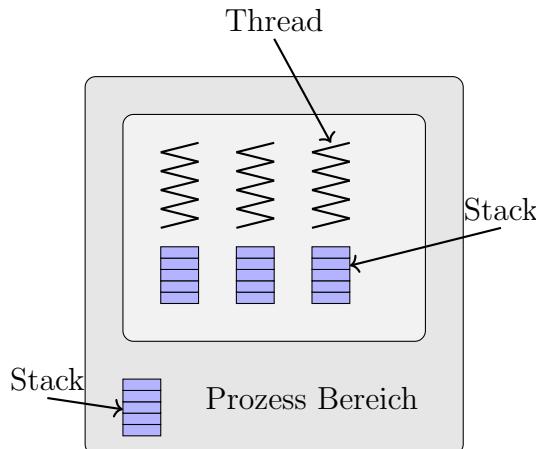


Abbildung 2.5: Threads

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** werden mithilfe einer Thread-library implementiert und sind unabhängig von dem Betriebssystem. Infolge können Nutzer Level Threads auch auf einem System ohne unterstütztem Multiprozessing in Betrieb genommen werden. Außerdem ist das Management von vielen Threads schneller, da keine Systemaufrufe getätigt werden müssen.
- **Kernel Level Threads** 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]

2.5 Threads in Java

In Java sind Threads auf dem Kernel Level implementiert. In Versionen vor 1.2(1998) gab es die so genannten grünen Threads. Diese waren Nutzer Level Threads und in der Java VM implementiert. Heutzutage nutzt Java Kernel Level Threads, welche den Vorteil haben die gesamte CPU zu nutzen. Ein Prozess mit mehreren Threads kann somit alle vorhandenen CPU Kerne benutzen. Im Modell der grünen Threads wurde dies über Umwege gelöst. So wurden die grünen Threads abwechselnd auf ein paar Kernel Level Threads verteilt, um mehrere Kerne nutzen zu können.[2]

2.5.1 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 getätigter werden können.[9]

```
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();  
    }  
}
```

Listing 2.2: mit Thread erweitern

Das Vorgehen hat seine Probleme. Denn es wird nicht die *run()* Methode von Thread überschrieben, sondern die von Runnable, welches von Thread implementiert wird. Wenn die Thread Klasse erweitert wird, kann man keine weiteren Klassen erweitern. Um ein solches Problem zu lösen, kann auf die zweite Möglichkeit zurückgegriffen werden.

implements Runnable: Bei dieser Methode wird nicht ein selbständiger Thread erschaffen. Das Interface Runnable hat nur eine Methode - die *run()* Methode.[10] Um einen Thread zu starten, muss ein Thread Object erschaffen werden, welche im Konstruktor ein Object des interfaces Runnable erwartet.

```
public class SimpleRunnable implements Runnable{
    public SimpleRunnable(){
    }

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

    public static void main( String [] args){
        SimpleRunnable r = new SimpleRunnable();
        Thread t = new Thread(r);
        t.start();
    }
}
```

Listing 2.3: Runnable implementieren

Der Vorteil dieses Vorgehen ist, dass Runnable leichtgewichtiger ist und erst bei Ausführung ein Thread Object erschaffen wird. Auch ist die Separierung zwischen "Aufgabe" und "Bearbeiter" sinnvoll. Man erhält eine große Flexibilität. Es können zusätzlich Einstellungen an dem erschaffenen Thread Object vorgenommen werden. Die Methoden wie *setPriority()* sind public.[9] Ein weiterer wichtiger Punkt ist, dass Runnables in einem Thread-Pool verwendet werden können.

2.5.2 Thread-Pool

Ein Thread-Pool ist wie eine zweite CPU nur in Software. Es gibt eine Warteschlange und eine festgelegte Anzahl an arbeitenden Threads. Der Unterschied ist, dass die Warteschlange nicht aus Threads besteht, sondern aus *Runnables*. Diese werden auch nur ein einziges Mal ausgeführt und dabei nicht unterbrochen. Auch wird die Warteschlange chronologisch abgearbeitet. Ein Thread-Pool hat somit ein *nicht-präemptives Scheduling*, welches mit dem *first in first out* Verfahren arbeitet.

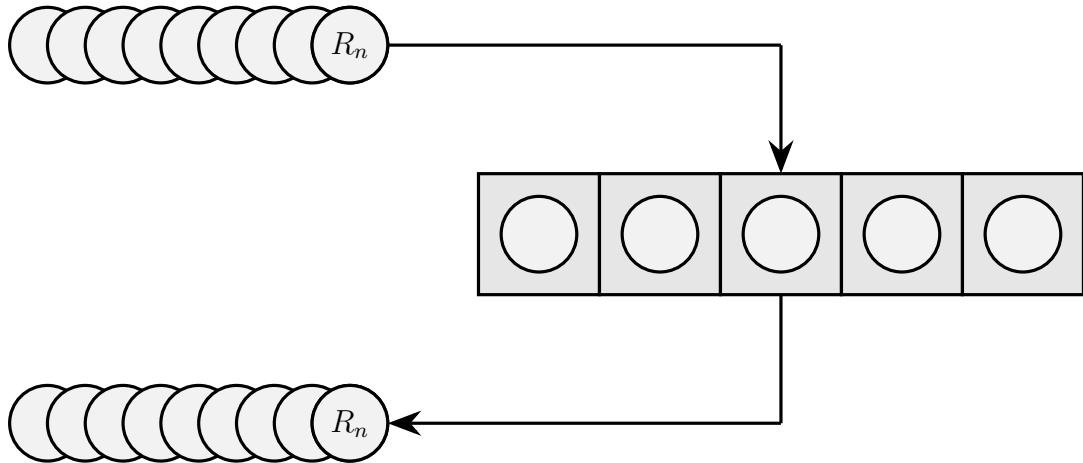


Abbildung 2.6: Thread-Pool²

In einem Thread-Pool werden die Aufgaben als *Runnable Object* verwaltet. Eine bestimmte Anzahl an Threads arbeiten und erhalten immer neue *Runnables*. Durch dieses Verfahren, muss nicht für jede Aufgabe ein neuer Thread erstellt werden. Auch ist ein Thread-Pool effizienter. Wenn viel berechnet werden muss, ist es besser weniger aktive Threads zu benutzen. Würden alle Aufgaben gleichzeitig gestartet werden, müsste der *System Scheduler* die Threads verwalten. Es würden somit viel mehr Kontextwechsel stattfinden, bis alle Aufgaben abgearbeitet sind. Daher ist eine optimale Größe eines Thread-Pools in der Größenordnung der vorhandenen Kerne.[13]

2.5.3 Threadsicherheit

[...]That the concept [of thread safety] itself is completely vague and essentially means nothing more than “behaves correctly in some situations” [...] - Eric Lippert [6]

Wie Herr Lippert in seinem Artikel bereits klarstellt, ist Threadsicherheit kein besonders kleines Thema. Auch ist Threadsicherheit immer im Auge des Betrachters. Es kommt immer auf die Situation an. Ist etwas in diesem speziellen Szenario Thread sicher? Denn eine allumfassende Threadsicherheit kann in den meisten Fällen nicht erreicht werden. Darum werde ich mich auf einen wichtigen Bereich der Threadsicherheit beschränken.

Wie in Kapitel 2.4 beschrieben, haben Threads einen gemeinsamen globalen Speicher. Das Problem dessen tritt auf, wenn mehrere Threads gleichzeitig auf

² Grafik in Anlehnung an https://en.wikipedia.org/wiki/Thread_pool

die selben Speicheradresse zugreifen. Ein simples Beispiel ist, dass erhöhen einer Zahl. Diese Situatuion tritt auch in meinem Programm, siehe Kapitel 3, auf. Dabei versuchen mehrere Threads den selben Integer zu erhöhen. Das Ergebnis ist, dass trotz 2000x erhöhen im Integer nur 1800 gespeichert ist. Um das Problem zu verstehen werde ich erklären, wie ein Integer erhöht wird.

Ein Integer wird nicht an seinem Speicherort geändert. Die Bytes werden erst in den Prozessor geladen und dann mit dem gewünschten Wert addiert. Der alte Wert wird mit dem neuen Wert aus dem Register überschrieben. Im Falle von vielen Threads fetcht³ sich ein Thread den Wert $W_1 = 500$. Thread zwei, welcher kurz nach Thread eins gestartet wurde, fetcht auch den Wert $W_2 = 500$. Beide addieren ihren Teil und erhalten $W_3 = 501$ und $W_4 = 501$. Beide Werte werden anschließend nacheinander auf den alten Wert geschrieben. Wurde somit der Wert zwei mal erhöht, ist im Ergebnis aber nur um eins gestiegen.[12]

Um solches Verhalten zu vermeiden, gibt es in Java das Schlüsselwort *synchronized*. Es beschränkt den Zugriff auf eine Methode oder Quelltext-Abschnitt auf nur einen Thread. Alle anderen Threads müssen auf das Beenden des vorherigen warten. Das Beispiel würde als Threadsichere Version wie folgt aussehen: [11]

```
public class ThreadSafe{  
    int counter=0;  
  
    public synchronized void increase(){  
        counter++;  
    }  
}
```

Listing 2.4: synchronized in Java

³ Deutsch: bringen - ein Programm ließt Daten.

3 Programm

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 entschieden, mit dem man ein gewünschtes Bild mit vielen weiteren Bildern rekonstruieren kann. Es wird demnach ein Mosaik aus Bildern erstellt.



(a) Eingabe



(b) Ausgabe

Abbildung 3.1: Funktionsweise des Programmes

Die Anwendung von Threads kommt in dem Programm an vielen Stellen vor. Im folgenden werde ich mich auf die Algorithmen der Bildanalyse und Verarbeitung beziehen. Andere Aspekte, wie die Implementation des Testmodus und anderen Funktionen, die in der App vorhanden sind, werden kurz im Anhang erwähnt.

3.1 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. Das 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 schrittweise und die nachfolgenden Kapitel stellen die Reihenfolge dar.

3.2 Implementation in Java

3.2.1 Berechnen der Sektoren

Die Größe des Bildes und in wie viele Teile dieses aufgeteilt werden soll ist bekannt. Dazu wird zum Beispiel die Breite des Bildes mit der horizontalen Anzahl an Sektoren geteilt. Ein Problem liegt darin, dass die Breite oder Höhe des Bildes nicht immer ein Vielfaches von den gewünschten Sektoren ist. Um das Problem zu lösen wird die Division mit Integers durchgeführt und der Rest gespeichert. Übergeblieben sind wie viele Spalten sich auf der X- und Y-Achse befinden und der Rest an Pixel. Im nächsten Schritt werden die Größen der Sektoren in Pixel berechnet. Es werden dazu zwei Integer Arrays für die X- und Y-Achse erschaffen. Jedes Element repräsentiert die Pixel Breite/Höhe seines Sektors. Standardmäßig besitzt jedes Element die Anzahl an Sektoren als Größe. Um den obigen berechneten Rest an Pixel zu verteilen, wird ein weiteres Array mit der Anzahl an Spalten als Länge aufsteigend nummeriert. Das Array wird anschließend von einer Methode gemischt. Eine abschließende Schleife wird mit der Anzahl des Restes wiederholt. Bei jedem Schritt wird ein dann zufälliger Wert aus dem 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 zweidimensionales Array an *BufferedImages* aus dem original Bild zu extrahieren. Dazu wird die *getSubimage()* Methode der Klasse *BufferedImage* verwendet.

3.2.2 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 beinhaltet. 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. Komprimierungsverfahren wie jpg und png kann die Dateigröße eines Bildes um das 10-fache verkleinern. Um die wirkliche Größe zu berechnen, müssen die Dimensionen und Farbtiefe verrechnet 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 entsprechen, 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 Integer definiert, welches beschreibt wie stark die nachfolgenden for-Schleifen bei jedem Schritt erhöht werden soll. Weiterhin gibt es drei Long. Diese speichern den gesamten rot, blau und grün Wert des Bildes. 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 Werten, geteilt durch die Anzahl der verwendeten Pixel, zurückgegeben.

3.2.3 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 das Vorgehen kann ein BinarySort Algorithmus auf 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 haben die Besonderheit, dass sie das *synchronized* Schlüsselwort besitzen (siehe Abschnitt 2.5.3).

Besonders werde ich auf die Methode *compareTo()*, des Interfaces Comparable eingehen. In der Methode werden zwei Farben verglichen. 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 einfacheren 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)	■
11111111	11100000	01000000	00010000	-2080752

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

3.2.4 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 Abschnitt 2.5). 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.

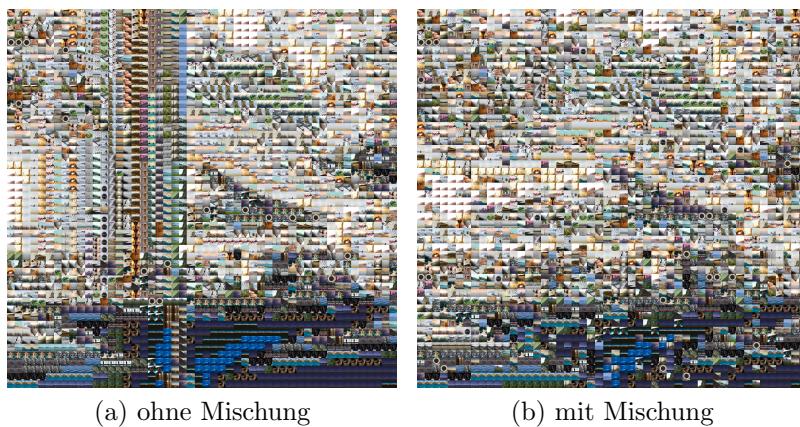


Abbildung 3.2: Unterschied zwischen mischen und nicht mischen

Jeder *Runnable* sucht demnach nach dem besten Bild, welches noch verfügbar ist. Aus jeder Datenbank wird mithilfe einer binären Suche das beste Bild bestimmt. Diese werden untereinander verglichen und das finale Bild bestimmt. Die binäre Suche wird mithilfe der Java Methode *Arrays.binarySearch* realisiert werden. Sie findet nicht nur den exakten Wert, sondern auch einen Index, wo er sich befinden könnte. Um das Verwendungslimit eines Bildes nicht zu überschreiten wird, von dem Index als Mitte, immer weiter nach links und rechts gegangen. Die beiden Werte, werden auf Farbnähe zum mittigen Wert überprüft. Der Nähere wird dementsprechend auserwählt. Durch die Nutzung der *fileAndColor* Klasse ist der Speicherort des Bildes bekannt.

3.2.5 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 wird in vier unterschiedlichen Größen verwendet werden. Die Größen entstehen aus dem in Abschnitt 3.2.1 beschriebenen Vorgehen. Werden die Bilder nur in einer einheitlichen Größe skaliert entstehen schwarze Linien im Bild.



Abbildung 3.3: vierfache Skalierung der Bilder

Beim Management 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 gefundenen Bildes zurückgegeben. Im Falle eines Treffers wird eine Referenz auf das Bild gespeichert, anstelle es zu skalieren.

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 sie arbeiten dürfen. Denn das Skalieren des Bildes und Speichern

dessen länger dauert als einen neuen Thread zu starten. Doch dadurch werden nicht nur Ressourcen unnötig verbraucht, sondern auch Fehler in der Bilddatei können durch gleichzeitiges zugreifen auf eine Datei entstehen.

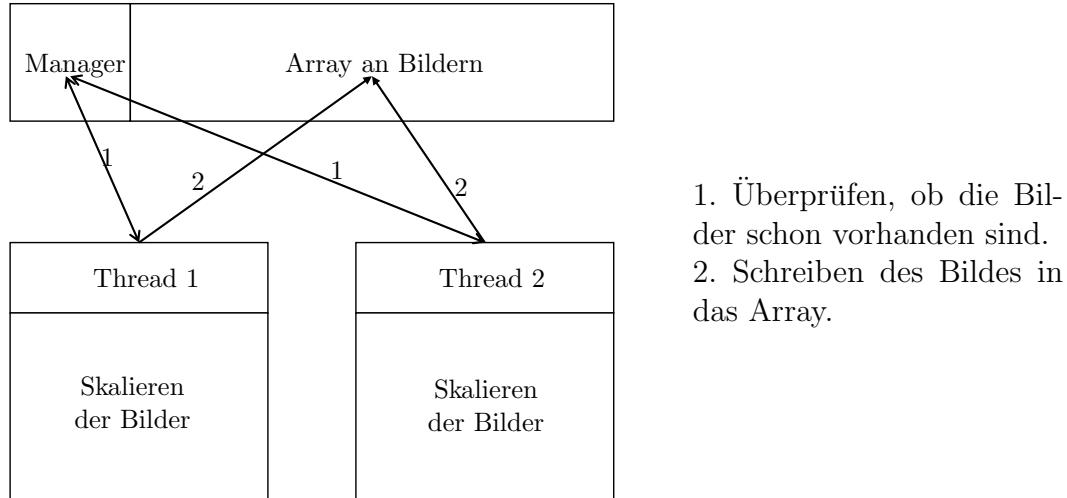


Abbildung 3.4: Threads mit Manager

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 nur auf den Index gezeigt wird und nicht auf dessen Inhalt. Die für gewöhnlich lineare Array Struktur wird durch rasantes steigen des Speicherverbrauchs einzelner Objekte nicht beeinflusst. Das liegt unter anderem an dem *GarbageCollector* von Java, welcher auch im Stande ist Arrays aus ihrem sonst linearem Besetzen eines Speicherblockes im Arbeitsspeicher aufzuteilen. Es kann dadurch größer werdenden Objekten Platz geschaffen werden. Interne Referenzen des Arrays werden 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 eines Bildes verwendet werden.

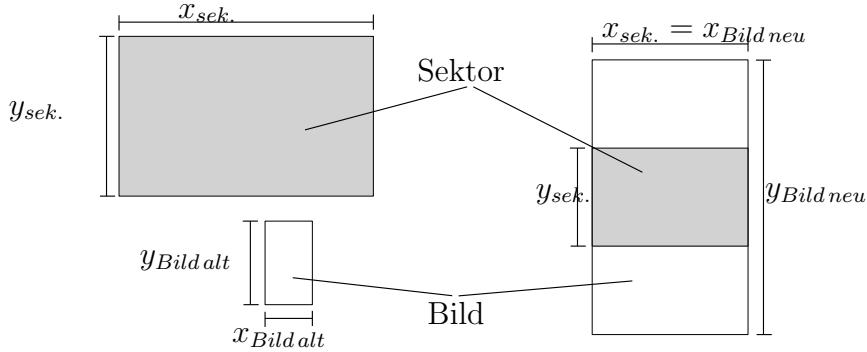


Abbildung 3.5: Berechnen Zielgröße

Um das erreichen zu können, wird das Bild erst auf die Größe des gewünschten Sektors skaliert. Anschließend werden die Ränder abgeschnitten. Beim Skalieren wird versucht die x-Achse des Bildes der y-Achse des Sektors anzupassen. Damit kann sichergestellt werden, dass das Bild sein 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 obige Vorgehen umgekehrt mit der x-Achse gemacht. Der letzte Schritt ist ein Abschnitt mittig aus dem skaliertem Bild mit den Abmaßen des Sektors zu schneiden.

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.[4] Im folgenden werde ich die Bilineare Interpolation erklären. Auf die bikubischen Interpolation 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 (nearest neighbour). Für Landschaftsbilder würde dies lediglich ein verpixeltes Bild in einer größeren Auflösung ergeben. Um neue Farbwerte zu berechnen, werden diese bei der bilinearen Interpolation linear zwischen den Pixeln ermittelt.

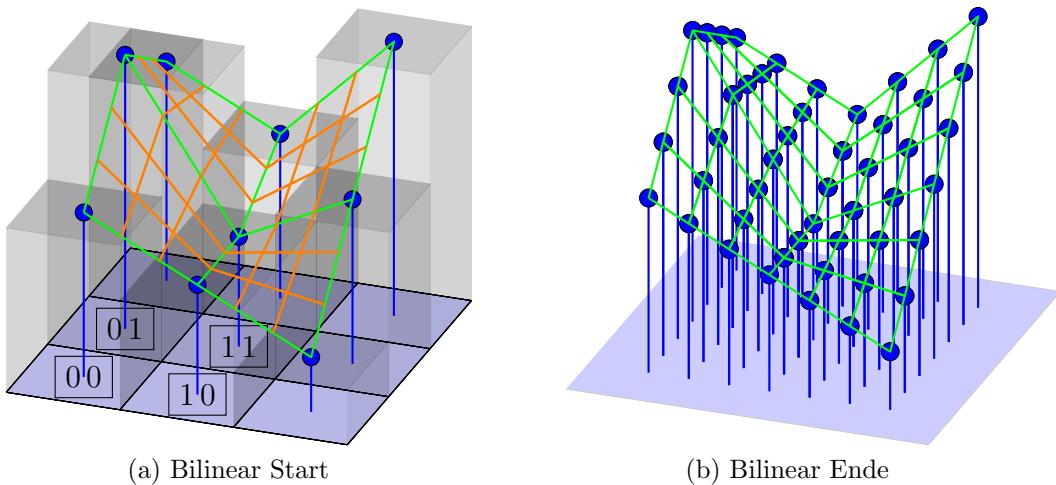


Abbildung 3.6: Bilineare Interpolation

Die unterschiedlichen Höhen in 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 den folgenden Gleichungen berechnen.[14, Ab 4:44]

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

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

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

In den Gleichungen 3.2 und 3.3 werden die Start- und End-Punkte einer Netzzlinie berechnet. n_x und n_y sind Zähler, welche die gewünschte Position in dem Teilnetz angeben. 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 und bikubischen Interpolation ist, dass bei der bilinearen Interpolation lediglich zwei Linien aus jeweils zwei Punkten benutzt werden. Bei der bikubischen Interpolation werden vier kubische Spline Funktionen aus jeweils vier Punkten benutzt. Der Vorteil der bikubischen Interpolation ist, dass das resultierende Bild einen höheren Kontrast bekommt. Auch können komplexere Formen besser berücksichtigt werden.[16, Seite 37]

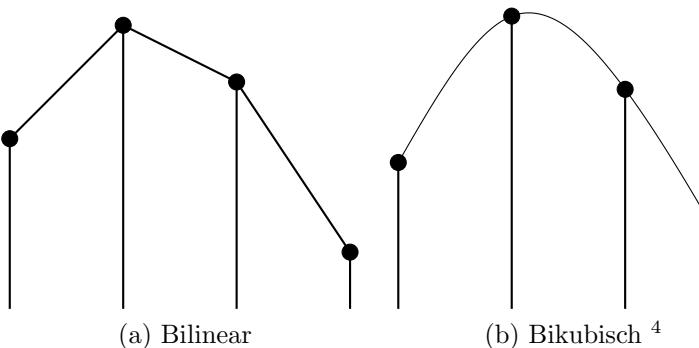


Abbildung 3.7: Vergleich Bilinear und Bikubisch

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.[4, Scalr.java Zeile 2221]

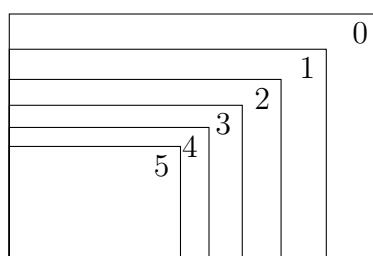


Abbildung 3.8: Inkrementelles Skalieren

⁴ Funktion mit <https://tools.timodenk.com/cubic-spline-interpolation> erstellt

4 Laufzeitanalyse

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 Berechnens 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 würde.

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

Die Laufzeit des Skalierens ist abhängig von dem genutzten Algorithmus.

Alle Tests wurden mit dem folgenden Computer durchgeführt.

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

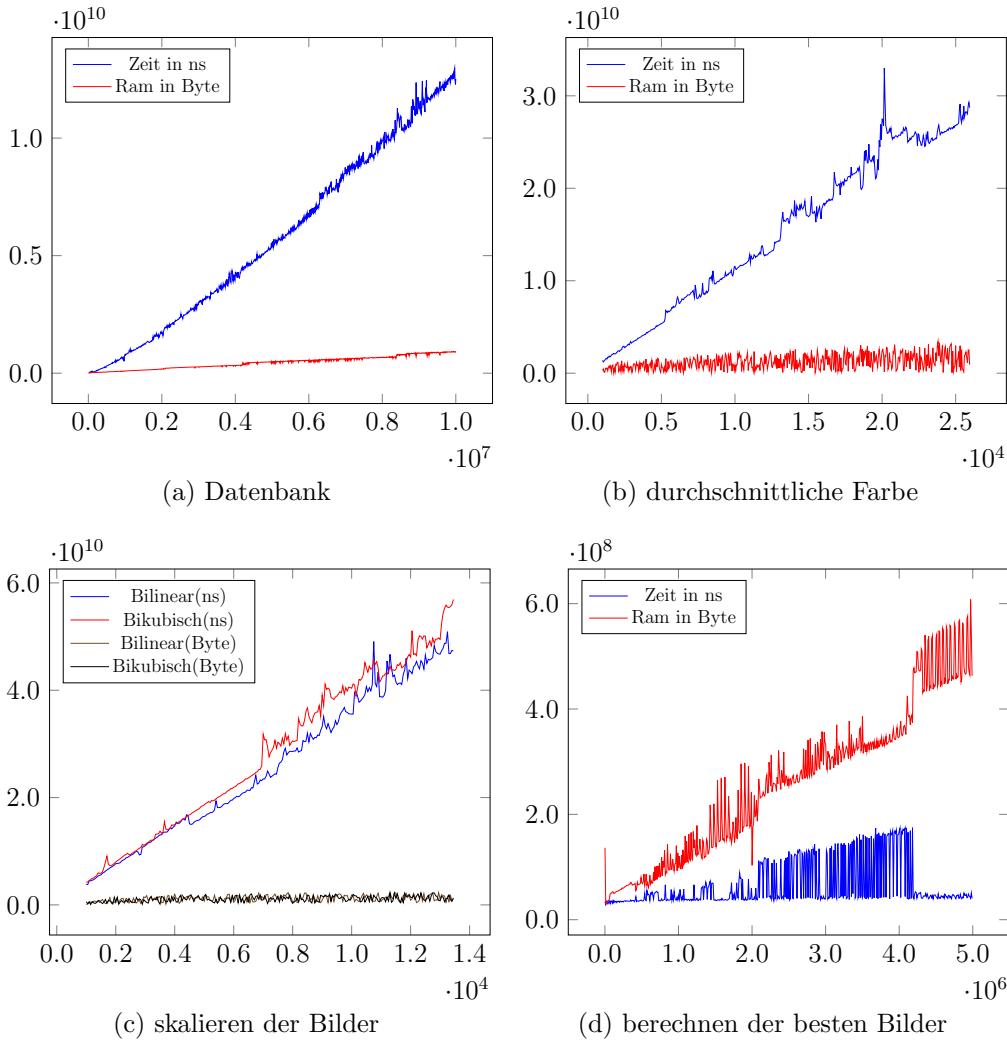


Abbildung 4.1: Vergleich Laufzeiten und Arbeitsspeicher

4.1 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 sortiertem Aufbau. Es 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 Bild⁵ verwenden. Ich werde die Datenbank nur einmal erstellen, da die Laufzeit von dieser sich nicht verändern 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 auf dem mitgegebenen Stick mit 50% Transparenz angesehen werden. Ich empfehle einen Computer mit 16GB Arbeitsspeicher, um sich das Bild anzusehen, da es 1GB als PNG verbraucht.

⁵ Bild von <https://media.jaguar.com/de-de/news/2020/11/jaguar-f-type-r-cabrio-siegt-bei-den-sport-auto-awards-2020>

5 Schluss

Das Bearbeiten dieser Arbeit hat mich vieles gelehrt. Nicht nur habe ich ein tieferes Verständnis für das Programmieren bekommen, sondern auch das Managen von großen Projekten⁶. Jedoch war der “Do it yourself” Weg nicht immer leicht und es war nicht immer einfach etwas richtiges zu finden. Besonders weil das Thema Prozesse nicht das Bekannteste im Internet ist. Meine Anfangs naive Einstellung “Threads sind nur da um mehr Kerne zu nutzen.” hat sich als Falsch herausgestellt. Die enorme Komplexität hinter den wohl wichtigsten Prinzipien für die moderne Welt sind nicht zu unterschätzen. Ich habe mich schließlich im Linux Kernel Code wiedergefunden, um zu verstehen wie ein *Scheduler* funktioniert. Aber auch das Erstellen der Arbeit mit L^AT_EX war herausfordernd, jedoch schlussendlich belohnend. Die Grafiken wurden mit *TikZ* erstellt.

5.1 Aussicht

In meinen zukünftigen Projekten möchte ich meine Nutzung von Threads weiterhin verbessern. Ich möchte gerne noch mehr Funktionen zu meinem Programm hinzufügen. Ein weiterer Meilenstein, den ich mir erarbeiten möchte, ist die Nutzung der Grafikkarte. Die enormen Rechenkapazitäten, welche in der Grafikkarte verborgen sind, würde mir neue Türen für noch größere Projekte öffnen. Denn die Grafikkarte setzt neue Dimensionen im Thema Multithreading.

⁶ Das Projekt betrug über 5540 Zeilen Code in 49 Klassen.

Literatur

- [1] BRAUN, C.: *Betriebssysteme kompakt*. Springer Berlin Heidelberg, 2020.
- [2] COX, G.: *Threading Models in Java*. <https://www.baeldung.com/java-threading-models>, 2021.
- [3] DUSEY, A.: *Threads and its types in Operating System*. <https://www.geeksforgeeks.org/threads-and-its-types-in-operating-system/>, 2021.
- [4] KALLA, R.: *imgscalr*. <https://github.com/rkalla/imgscalr>, 2018.
- [5] KERRISK, M.: *sched - overview of CPU scheduling*. <https://man7.org/linux/man-pages/man7/sched.7.html>, 2021.
- [6] LIPPERT, E.: *What is this thing you call thread safe?*. <https://docs.microsoft.com/en-us/archive/blogs/ericlippert/what-is-this-thing-you-call-thread-safe>, 2009.
- [7] LOHMANN, D.: *Der Scheduler von Windows*. https://www4.cs.fau.de/Lehre/WS06/V_BS/folien/09-Sched-Windows-2x2.pdf, 2006.
- [8] LOVE, R.: *What is a tickless kernel?*. <https://www.quora.com/What-is-a-tickless-kernel>, 2013.
- [9] ORACLE: *Class Thread*. <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>.
- [10] ORACLE: *Interface Runnable*. <https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html>.
- [11] ORACLE: *Synchronized Methods*. <https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>, 2014.
- [12] ORACLE: *Thread Interference*. <https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html>, 2014.
- [13] ORACLE: *Thread Pools*. <https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html>, 2014.
- [14] POUND, D. M.: *Resizing Images - Computerphile*. https://youtu.be/AqscP7rc8_M?t=225, 2016.
- [15] REBEIRO, P. C.: *Lecture 22 Completely Fair Scheduling*. <https://www.youtube.com/watch?v=scfD0of9pww>, 2017.

- [16] ROWE, D. B.: *BiLinear, Bicubic, and In Between Spline Interpolation*. https://www.mssc.mu.edu/~daniel/pubs/RoweTalkMSCS_BiCubic.pdf, 2018.
- [17] TORVALDS, L.: *Linux Documentation*. <https://github.com/torvalds/linux/tree/master/Documentation/scheduler>, 2022.
- [18] TORVALDS, L.: *Linux Kernel*. <https://github.com/torvalds/linux/tree/master/kernel/sched>, 2022.
- [19] 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.
- [20] WILLIAMS, L.: *Process vs Thread: What's the difference?*. <https://www.guru99.com/difference-between-process-and-thread.html>, 2022.

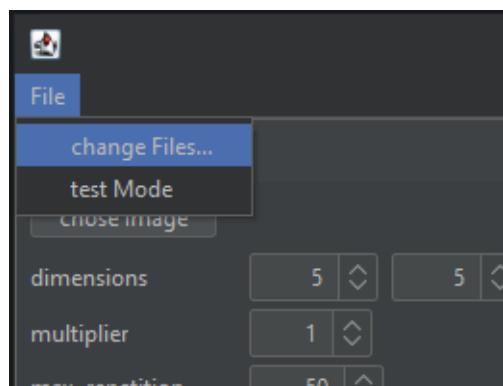
Hiermit erkläre ich, Philipp Bleimund, dass ich die Arbeit ohne fremde Hilfe und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Bielefeld, 06.04.2022

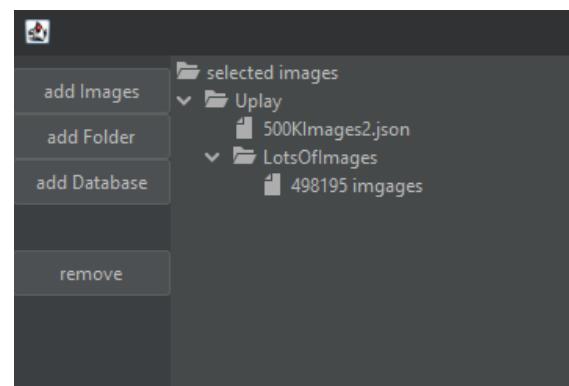
Programmbedienung

Auswählen von Dateien

Um Dateien auswählen zu können, muss im Hauptbereich auf File → change Files... gedrückt werden. Es öffnet sich ein neues Fenster. In diesem können Bilder, Ordner oder eine Datenbank ausgewählt werden. Ordner auswählen wird empfohlen, wenn viele Unterordner genutzt werden, oder die ausgewählten Bilder eine Anzahl von 50.000 überschreitet. Die Ordnerstruktur wird im rechten Bereich dargestellt, wobei alle Oberordner, welche keine Ausgewählten Bilder beinhalten. Ein Ordner und all seine Unterordner, können gelöscht werden. Beim verlassen des Fensters werden die Änderungen nur dann übernommen, wenn auf Speichern gedrückt wird.



(a) Dateibereich öffnen



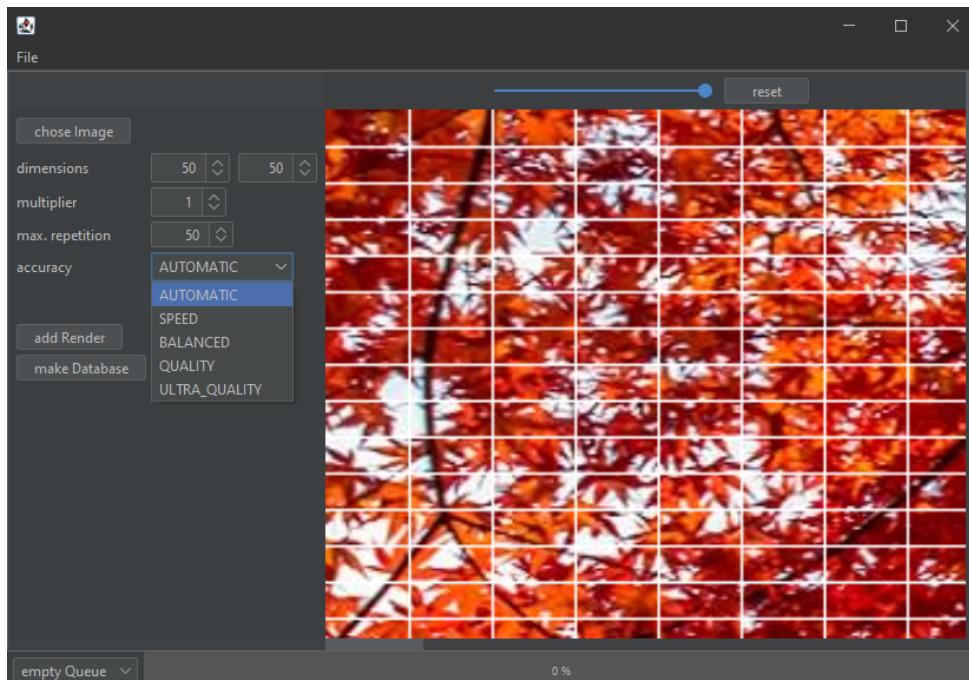
(b) Dateien auswählen

Ausschnitt 1: Dateien auswählen

Vorbereiten eines Render-Auftrages

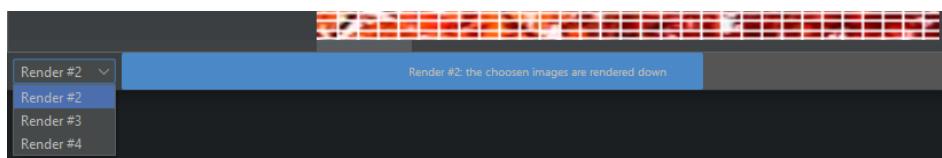
Um einen Auftrag vorzubereiten, kann man ein Grundbild auswählen. Die Spalten auf X und Y Ebene kann unter dem Bereich “Dimensions” angepasst werden. Es wird im Bildbereich auf der rechten Seite ein Netz aus Linien angezeigt, welche die Größe der Sektoren darstellt. Im Bereich “multiplier” kann angegeben werden, um welchen Faktor das Endbild skaliert werden soll. Es werden nur ganze Zahlen akzeptiert. Der Bereich “max. repetition” gibt an, wie häufig ein einzelnes Bild verwendet werden darf. Die Qualität von der Berechnung der durchschnittlichen Farbe und der Skalierung kann in einem Dropdown Menü ausgewählt werden.

Der Auftrag kann abgeschickt werden, oder alternativ auch eine Datenbank mit der ausgewählten Quantität erstellt werden kann.



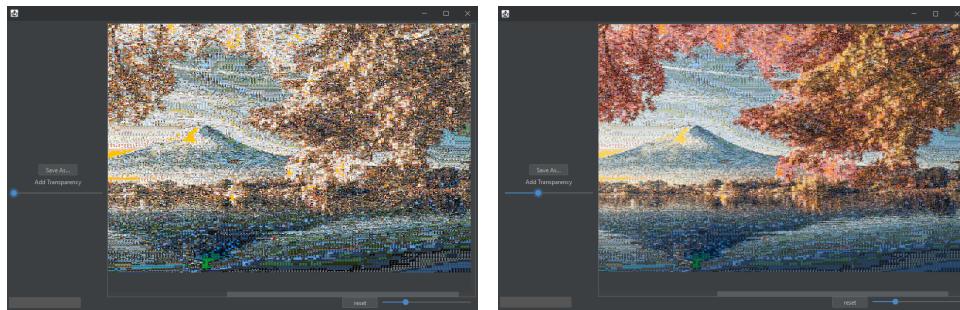
Ausschnitt 2: Hauptbereich

Es können auch mehrere Aufträge abgeschickt werden. Es wird jedoch nur einer bearbeitet. Zusätzlich wird der aktuelle Progress angegeben.



Bearbeiten des fertigen Bildes

Ist der Auftrag abgeschlossen, öffnet sich das fertige Bild in einem neuen Fenster. In diesem kann das Bild vor dem Speichern noch bearbeitet werden. Das original Bild kann mithilfe eines Schiebereglers mit unterschiedlicher Transparenz auf das Bild gelegt werden.



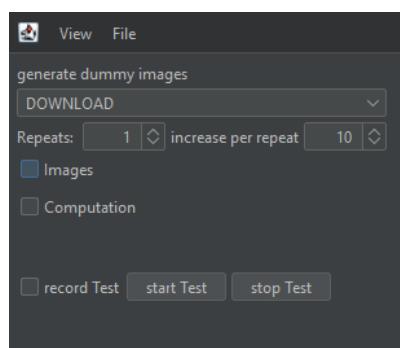
(a) ohne Transparenz

(b) mit Transparenz

Ausschnitt 3: Bildbearbeitung

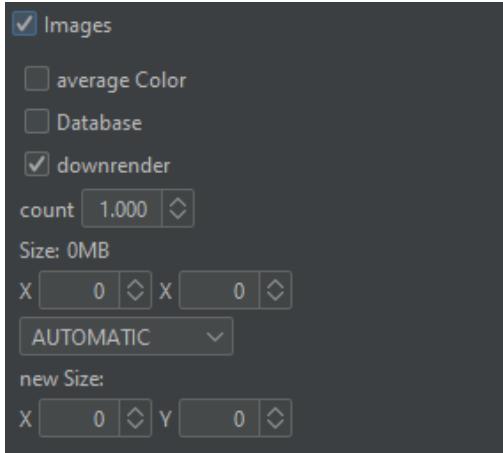
Nutzung des Testbereiches

Der Testbereich kann aus dem Hauptbereich erreicht werden, wenn in dem Menü auf Files → test Mode geklickt wird. Es wird sich ein neues Fenster öffnen. In dem Fenster kann zwischen den unterschiedlichen Test Modi ausgewählt werden. Auch können spezifische Einstellungen vorgenommen werden. Dabei kann man auswählen aus welcher Quelle die verwendeten Bilder generiert werden. Zur Auswahl stehen generierte Bilder und Bilder, welche aus dem Internet heruntergeladen werden. Die Wiederholrate der Tests kann auch spezifiziert werden. Bei jeder Wiederholung wird die Anzahl der Bilder erhöht. Die Anzahl an Bildern kann im rechten Feld eingegeben werden. Außerdem gibt es die Möglichkeit den Test aufzuzeichnen. Ein bereits laufender Test kann abgebrochen werden. Dies ist aber als Not-Maßnahme gedacht. Es wird empfohlen das Programm danach neu zu starten.

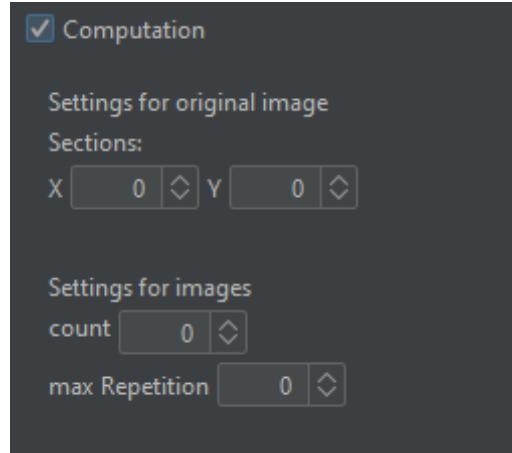


Ausschnitt 4: Testbereich

Die unterschiedlichen Test Szenarien sind: Die durchschnittliche Farbe berechnen, eine Datenbank erstellen, Bilder skalieren und die besten Bilder berechnen. In dem Bilder Sektor, werden alle nötigen Einstellungen angezeigt. Diese variieren nach Auswahl.



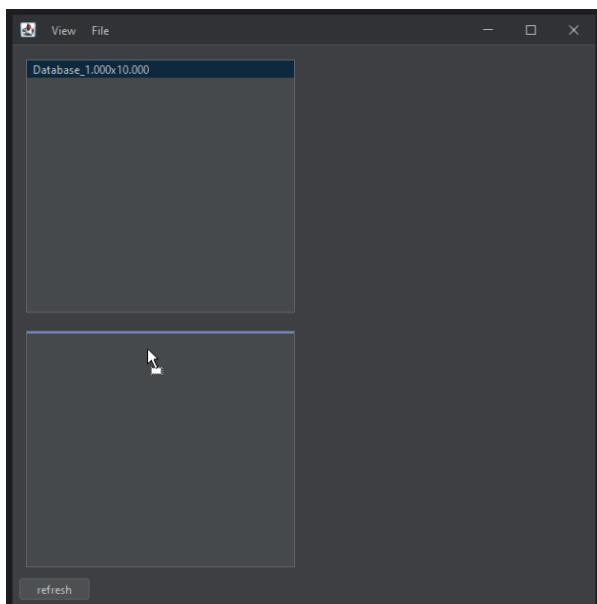
(a) Bilder



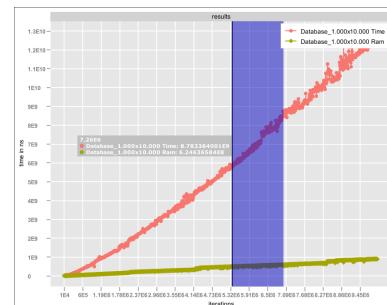
(b) Berechnung

Ausschnitt 5: Test Einstellungen

Wurde ein Test durchgeführt, so können seine Ergebnisse angesehen werden. Dazu muss auf view →select view →graph gegangen werden. Es erscheinen auf der linken Seite zwei Bereiche. Im oberen stehen alle gemachten Tests. Im unteren Bereich stehen alle Tests, die man ansehen möchte. Um einen Test anzusehen, muss dieser einfach von oben nach unten mit gedrückter Maus gezogen werden.



(a) Tests auswählen



(b) Graph betrachten

Ausschnitt 6: Tests ansehen