Werkstück A mit dem Thema dynamische Partitionierung der Portfolioprüfung des Moduls Betriebssysteme und Rechnernetze im Sommersemester 2022

Steffen Rottke und Michelle Müller

Frankfurt University of Applied Sciences

Das Werkstück befasst sich mit der Umsetzung der Entwicklung eines Simulators zur Darstellung unterschiedlicher Realisierungskonzepte für die dynamische Partitionierung. Die Umsetzung in Form eines Bash-Skriptes erfordert drei wesentliche Bereiche. Diese sind die Steuerung des Simulators, die einzelnen Realisierungskonzepte wie First Fit, Next Fit, Best Fit, Worst Fit, Last Fit und Random sowie die visuelle und informative Darstellung zu den angewendeten Verfahren. Zudem beinhaltet ist die Zusammenführung und Implementierung der einzelnen Programmteile mit Hilfe der Plattform GitHub.

Die Dokumentation über die Erstellung des Simulators ist in drei Teile unterteilt. Der erste Teil beginnt mit der Steuerung des Simulators, die ebenfalls den Beginn des Programmes darstellt. Daraufhin folgen die Erläuterungen zu den verschiedenen Konzepten, die die jeweiligen Benutzereingaben verarbeiten und anschließend folgt die Dokumentation zur Erstellung der Darstellung der Speicherbelegung, die der Benutzer schlussendlich ausgegeben bekommt.

Die Aufgabe zur Umsetzung eines Simulators, der verschiedene Realisierungskonzepte visuell darstellen kann, erfordert einige Schritte bis zur letztendlichen Realisierung. Neben der jeweiligen Programmierung der Abfragen und Steuerung der Simulation, der Abläufe der einzelnen Realisierungskonzepte sowie der Darstellungsform, ist eine Herausforderung die Schnittstelle der Übertragung der eingegebenen Daten durch den Benutzer bei der Hauptsteuerung zu dem jeweilig gewählten Realisierungskonzept. Des Weiteren benötigt die Umsetzung des Simulators die Erstellung einer Schnittstelle zwischen dem jeweiligen Realisierungskonzept mit den übergebenen Daten und der Ausgabe der jeweiligen Speicherbelegung in visueller und informativer Darstellungsform, welche wiederum die Hauptsteuerung aufruft.

Die Kommunikation

Der gruppeninterne Austausch über die Projektdateien erfolgte über das Versionsverwaltungssystem für Software GitHub [1]. Die Nutzung der GitHub-Funktion Branches ermöglichte die reibungslose Arbeit im Team und erleichterte das Kommunizieren zwischen den einzelnen Teammitgliedern. Aufgrund dessen, dass die aktuellste Version der jeweiligen Dateien abrufbar war, wurde ebenfalls ein individuelles und aktuelles Arbeiten möglich.

Die Steuerung des Simulators

Zunächst einmal ist die Steuerung des Simulators in drei Komponenten unterteilt. Die erste ist die Hauptsteuerung, die zweite die Simulationssteuerung und die dritte Komponente sind die Funktionen der Simulationssteuerung.

Die Hauptsteuerung beinhaltet die Befehle, die die jeweiligen Funktionen oder Dateien aufrufen sowie das Gerüst der Abfolge der Steuerung, wie die Frage nach Benutzereingaben zur Gesamtgröße oder dem Realisierungskonzept. Die Simulationssteuerung hingegen beinhaltet die Befehle rund um die Erstellung, Löschung oder das Zurücksetzen der Partitionen bzw. der Partitionsbelegung. Die dritte Komponente sind die Funktionen für die Simulationssteuerung auf die sowohl die Hauptsteuerung als auch die Simulationssteuerung zugreifen, um einen reibungslosen Programmablauf sicherzustellen.

Die Hauptsteuerung

Zu Beginn der Datei *hauptsteuerung.sh*, die die Hauptsteuerung darstellt, befindet sich die Abfrage der Gesamtspeichergröße, welche vom Benutzer festgelegt werden soll (siehe Abbildung 1). Dieser kann daraufhin die Speichergröße eingeben, welche danach weiterverarbeitet wird. Dafür wurde zu Beginn der Datei die Quelle angegeben, diese ist *simulationssteuerungFunktionen.sh*, die benötigt wird, um die im Folgenden verwendeten Funktionen aufzurufen. Mit Hilfe einer while-Schleife wird überprüft, ob die Eingabe weitere Zeichen außer Zahlen enthält und bzw. oder kleiner als null ist. Wenn dies der Fall sein sollte wird erneut nach einer Eingabe gefragt. Sobald die Eingabe die beiden Kriterien erfüllt, dass keine Zeichen enthalten sind und sie auch größer als null ist, übergibt der Code die Eingabe, an die Variable *gewaehlteGesamtspeichergroesse*. Dieses Verfahren zur Eingabeaufforderung und zur Überprüfung nach gewissen Kriterien ist im folgenden Code mehrfach angewandt.

```
#Benutzer wird aufgefordert, eine Gesamtspeichergroesse zu waehlen
echo "Bitte geben Sie eine Gesamtspeichergroesse an."
#Speichergroesse wird eingelesen
read -r speichergroesse
        #Nachdem die Groesse eingelesen wurde wird ueberprueft, ob die Eingabe andere Zeichen ausser Zahlen enthaelt oder nicht groesser als 0 ist
        #Wenn das der Fall ist, wird solange nach der Groesse gefragt, bis keine anderen Zeichen ausser Zahlen enthalten sind und die Eingabe groesser 0 ist
        #Als Quelle fuer die Funktionsaufrufe wird simulationssteuerungFunktionen.sh genutzt
        #Der Rueckgabewert der Funktion keineZeichenGroesserNull ist die Gesamtspeichergroesse, welche in der Variablen gewaehlteGesamtspeichergroesse gespeichert wird
        while [ -n "$(printf '%s\n' "$speichergroesse" | sed 's/[0-9]//g')" ] || [[ ! $speichergroesse -gt 0 ]]
        do
                keineZeichenNichtGroesserNull speichergroesse
                zeichen speichergroesse
                read -r speichergroesse
        done
        gewaehlteGesamtspeichergroesse=$(keineZeichenGroesserNull)
echo "Ihre gewaehlte Speichergroesse ist: $gewaehlteGesamtspeichergroesse"
```

Abbildung 1. Code-Ausschnitt der Abfrage zur Gesamtspeichergröße

Das nächste Beispiel dafür ist die Abfrage nach dem Realisierungskonzept (siehe Abbildung 2).

```
#Benutzer wird aufgefordert, das Realisierungskonzept auszuwaehlen, es gibt die Wahl zwischen den Shortcuts f,b,n,w,r und l
echo "Bitte geben Sie ein, welches Realisierungskonzept genutzt werden soll. Es stehen folgende Realisierungskonzepte zur Verfuegung:
-> f (First Fit)
-> b (Best Fit)
-> n (Next Fit)
-> w (Worst Fit)
-> r (Random)
-> l (Last Fit)"
#Das ausgewaehlte Konzept wird eingelesen
read -r konzept
        #Als Quelle wird die Datei simulationssteuerungFunktionen.sh genutzt
        #Es wird ueberprueft, ob die Eingabe einem der Realisierungskonzepte entspricht oder ob erneut nach einer Eingabe gefragt wird
        while ! { [ "$konzept" = "f" ] || [ "$konzept" = "b" ] || [ "$konzept" = "n" ] || [ "$konzept" = "w" ] || [ "$konzept" = "r" ] || [ "$konzept" = "l" ]; }
        do
                ungueltigesRealisierungskonzept konzept
                read -r konzept
        done
```

Abbildung 2. Code-Ausschnitt der Abfrage zum Realisierungskonzept

In diesem Fall ist das Kriterium der while-Schleife, dass die Benutzereingabe entweder ein f, b, n, w, r oder l darstellt, wobei der jeweilige Buchstabe für eines der Realisierungskonzepte steht.

Außerdem wird die Eingabe direkt ein weiteres Mal überprüft, um zu entscheiden, welche Befehle im Folgenden auszuführen sind. Als Beispiel dient hierfür der Code für das Realisierungskonzept First Fit (siehe Abbildung 3).

```
case "$konzept" in
"f")
        echo "Ihr gewaehltes Konzept ist: First Fit."
        #Die Funktion abfrageAktion wird von der Quelle simulationssteuerungFunktionen.sh aufgerufen
        abfrageAktion
        #Der Befehl wird eingelesen
        #Es wird die Funktion ablaufRealisierungskonzept aufgerufen und der eingelesene Befehl uebergeben
        #Nachdem der jeweilige Befehl ausgewaehlt wurde (alle moeglich ausser q), wird die Funktion insertFirstFit aufgerufen, wenn der Befehl ein c ist
        #Der Funktion werden dabei drei Werte uebergeben
        #Nachdem die Funktion ausgefuehrt wurde, wird erneut nach einer Eingabe gefragt
        read -r befehl
                while [ "$befehl" != "q" ]
                do
                        ablaufRealisierungskonzept befehl
                        if [ "$befehl" = "c" ]; then
                        insertFirstFit $gewaehlteGesamtspeichergroesse "$createPartitionsName" $createPartitionsGroesse
                        fi
                        #Die Speicherbelegung wird nun ausgegeben
                        echo ""
                        echo "Die Speicherbelegung sieht wie folgt aus: "
                        #Visuelle Ausgabe des Speichers und der Informationen zur Speicherbelegung durch den Funktionsaufruf von visualout aus der Datei visualout.sh
                        visualout
                        abfrageAktion
                        read -r befehl
                done
        #Wenn der Befehl quit gewaehlt wurde, wird nun ausgegeben, dass der Simulator beendet wurde und im Hintergrund wird die Speicherbelegung zurueckgesetzt
        echo "Der Simulator wurde beendet."
        ;;
```

Abbildung 3. Code-Ausschnitt der Überprüfung und Ausführung des Realisierungskonzeptes *First Fit*

Zu Beginn wird geprüft, ob die Benutzereingabe einem *f* für First Fit entspricht, da dadurch im nächsten Schritt die Funktion *abfrageAktion* aufgerufen wird, die den Benutzer nach einer Wahl des nächsten Befehls fragt. Diese können dann entweder *create*, *delete* oder *new* sein. Sobald auch diese Eingabe getätigt und überprüft wurde, dass die Eingabe kein *q* ist, wird das jeweilige Realisierungskonzept, in diesem Fall First Fit ausgeführt, die Ergebnisse, in Form von Rückgabewerten, an die jeweilige Datei des Realisierungskonzeptes übergeben und erneut nach einer Eingabe gefragt, solange es kein *q* ist. Sobald das *q* eingegeben wurde, wird die while-Schleife verlassen und der Simulator beendet.

Die Simulationssteuerung

Die Datei simulationssteuerung.sh, enthält zu Anfang die Prüfung, welcher Eingabebefehl durch den Datei- und Funktionsaufruf an die Funktion *simulationssteueung* übergeben wurde. Als Beispiel wird hier das *c* für den Befehl *create* betrachtet, um eine Partition zu erstellen (siehe Abbildung 4). Als erster Schritt kommt die Frage nach dem Partitionsnamen und der Partitionsgröße. Nichtsdestotrotz erfolgen auch bei diesen Benutzereingabeaufforderungen Prüfungen. Die Eingabe des Namens wird darauf überprüft, ob sie mit einem Leerzeichen und einem darauffolgenden c beendet wurde, wenn nicht, erfolgt wieder eine erneute Aufforderung zur Eingabe. Die gültige Eingabe wird wiederum in der Variablen *createPartitionsName* gespeichert. Des Weiteren wird nach der dazugehörigen Partitionsgröße gefragt. Diese darf weder andere Zeichen außer Zahlen beinhalten, muss mit einem c bestätigt sein und muss größer null sein. Sonst erfolgt eine erneute Eingabeaufforderung mittels der while-Schleife. Die Größe ist in der Variablen *createPartitionsGroesse* gespeichert.

```
case  "$befehl" in
"c")
        #Wenn der Befehl create entspricht, wird nun die Abfrage nach dem Namen und anschliessend nach der Groesse gestartet
        #Die Variablen createPartitionsName und createPartitionsGroesse speichern den Namen bzw. die jeweilige Groesse bis zu dem eingegebenen c
        echo 'Bitte geben Sie einen Namen für die Partition ein und bestaetigen Sie mit: " c".'
                read -r name
                        #Nachdem der Name eingelesen wurde wird ueberprueft, ob die Einagbe mit einem c bestaetigt wurde oder nicht
                        #Wenn nicht, wird solange nach dem Namen gefragt, bis mit c bestaetigt wurde
                        #Die Funktionen zur Ueberpruefung befinden sich in simulationssteuerungFunktionen.sh
                                while [[ ! "$name" =~ " c" ]]
                                do
                                        frageNachNameNichtBestaetigt name
                                        read -r name
                                done
                                createPartitionsName=$(frageNachNameBestaetigt)
                                #Die Funktion pruefenAufGleichenNamen ueberprueft, dass eine erstellte Partition nicht den gleichen Namen hat wie eine bereits erstellte
                                pruefenAufGleichenNamen $(frageNachNameBestaetigt)
        echo 'Bitte geben Sie eine Groesse für die Partition ein und bestaetigen Sie mit: " c".'
                        read -r groesse
                        #Nachdem die Groesse eingelesen wurde wird ueberprueft, ob die Eingabe mit einem c bestaetigt wurde oder nicht und ob es sich um eine Zahl groesser$
                        #Wenn nicht, wird solange nach der Groesse gefragt, bis mit c bestaetigt wurde und die Eingabe nur Zahlen groesser 0 enthaelt
                        #Die Funktionen zur Ueberpruefung befinden sich in simulationssteuerungFunktionen.sh
                                while [ -n "$(printf '%s\n' "${groesse/ c*/}" | sed 's/[0-9]//g')" ] || [[ ! "$groesse" =~ " c" ]] || [[ ! ${groesse/ c*/} -gt 0 ]]
                                do
                                        zeichenBestaetigt groesse
                                        zeichenNichtBestaetigt groesse
                                        keineZeichenBestaetigtNichtGroesserNull groesse
                                        keineZeichenBestaetigtGroesserNull groesse
                                        keineZeichenNichtBestaetigtNichtGroesserNull groesse
                                        keineZeichenNichtBestaetigtGroesserNull groesse
                                        read -r groesse
                                done
                        createPartitionsGroesse=$(keineZeichenBestaetigtGroesserNull)
        echo "Der eingegebene Name für die Partition ist: $createPartitionsName und die gewaehlte Groesse ist: $createPartitionsGroesse."
        ;;
```

Abbildung 4. Code-Ausschnitt zum Erstellen einer neuen Partition

Eine weitere Möglichkeit ist, die Speicherbelegung zurückzusetzen mit dem Befehl *n* für *new* (siehe Abbildung 5).

```
"n")
        #Die Datei simulationssteuerungFunktionen wird aufgerufen, um auf die Funktion speicherZuruecksetzen zuzugreifen, die die derzeitigen Namen, die Groessen und den S$
        speicherZuruecksetzen save_name save_groesse save_speicher
        ;;
```

Abbildung 5. Code-Ausschnitt zum Zurücksetzen der Speicherbelegung

Hierfür wird die Funktion *speicherZuruecksetzen* zum Zurücksetzen des Speichers aufgerufen, die alle bisher erstellten und noch existierenden Partitionen löscht und die verfügbare Gesamtspeichergröße dadurch wieder komplett zurücksetzt.

Alle anderen eingegebenen Befehle führen zu einer Fehlermeldung und zur Darstellung des Speichers sowie den dazugehörigen Informationen durch den Dateiaufruf für die visuelle und informative Darstellung. Ebenfalls führen alle erfolgreich ausgeführten Befehle zu der Darstellung des Speichers und der Ausgabe der dazugehörigen Informationen.

Die Funktionen für die Simulations- und Hauptsteuerung

Die Datei, die die Funktionen zur Steuerung beinhaltet ist *simulationssteuerungFunktionen.sh*. Darin befinden sich alle Funktionen zur Überprüfung, ob gegebenenfalls andere Zeichen außer Zahlen in den Eingaben enthalten sind, ob sie mit dem Befehl für *create* (*c*) bestätigt wurden und bzw. oder ob sie größer als null sind. Ein Beispiel ist die Funktion *keineZeichenBestaetigtGroesserNull* (siehe Abbildung 6), die ausgeführt wird, wenn der Benutzer nur Zahlen verwendet hat, die Eingabe größer null ist und diese Eingabe auch mit einem c bestätigt hat.

```
#Funktion fuer die Partitionen zum Ueberpruefen, dass keine Zeichen in der Eingabe vorhanden sind, die Eingabe mit c bestaetigt wurde und die Eingabe groesser 0 ist
keineZeichenBestaetigtGroesserNull () {
        if [ ! -n "$(printf '%s\n' "${groesse/ c*/}" | sed 's/[0-9]//g')" ] && [[ "$groesse" =~ " c" ]] && [[ ${groesse/ c*/} -gt 0 ]]; then
                echo ${groesse/ c*/}
        fi
}
```

Abbildung 6. Code-Ausschnitt der Funktion, bei der die Eingabe keine Zeichen enthält, mit c bestätigt wurde und größer als null ist

Des Weiteren gibt es Funktionen, die überprüfen, ob der eingegebene Buchstabe einem der vorgegebenen Auswahlmöglichkeiten für das jeweilige Realisierungskonzept entspricht, ob der eingegebene Befehl kein *q* war, damit die Realisierungskonzepte und die dafür notwendigen Eingaben weiterhin stattfinden oder eine Funktion, die beispielsweise kontrolliert, dass keine Namen für Partitionen doppelt vergeben werden. Weitere Funktionen werden aufgerufen, wenn der Benutzer nach einer Eingabe zu dem jeweiligen Befehl für die Simulationssteuerung gefragt werden soll und auch die Funktion, um die Speicherbelegung komplett zurückzusetzen befindet sich in dieser Datei.

Die Realisierungskonzepte

Die verschiedenen Realisierungskonzepte stammen sowohl von den gestellten Anforderungen, als auch aus eigener Initiative. Einige von Ihnen sind in der echten Welt eher von Vorteil als andere, allerdings dienen alle sehr gut der Visualisierung von Effizienz und Ablauf verschiedener Algorithmen.

Um die Abläufe besser gegenüber stellen zu können, fokussiert sich die Dokumentation auf First-Fit und Last-Fit sowie Best-Fit und Worst-Fit. Es gibt zusätzlich noch Random-Fit und Next-Fit.

First-Fit und Last-Fit

Die beiden Verfahren First-Fit (siehe Abbildung 7) und Last-Fit (siehe Abbildung 8) sind an sich sehr ähnliche Konzepte, nur durchläuft First-Fit wie der Name schon verrät den Speicher von vorne und Last-Fit von hinten.

Aufgrund dessen wird der Speicher in zwei entgegengesetzte Richtungen befüllt.

```
seekCounter=0 #Counter der die freien Plätze beinhaltet
for (( i=0; i<=$save_speicherplatz; i++ )) #Schleife in der Größe des Speicherplatzes
do
---->if [[ -z "${save_speicher[$i]}" ]]; then #Wenn die i-te Stelle im Speicher leer ist
---->    seekCounter=$(($seekCounter+1)) #Aufeinander folgende leere Plätze werden hochgezählt
---->    if [[ $seekCounter -ge $seekGroesse ]]; then #Wenn der Counter der leeren Plätze größer/gleich der zu Speichernden Größe ist
---><------>index=$(($i-($seekCounter-1))) #Startposition festlegen
---><------>for (( y=$index; y<$(($index+$seekGroesse)); y++ )) #Die Schleife X-Mal durchlaufen (größenabhängig)
---><------>do
---><------>    save_speicher[$y]=$seekGroesse #$seekGroesse in das Array einfügen
---><------>done
---><------>save_groesse[$index]=$seekGroesse #Größe an der Stelle $index im Array einfügen
---><------>save_name[$index]=$seekName #Name an der Stelle $index im Array einfügen
---><------>break
---->    fi
---->else
---->    seekCounter=0 #Counter resetten
---->fi
done
```

Abbildung 7. Logik eines Ausschnittes von dem Algorithmus für das First-Fit Verfahren

Das Konzept Last-Fit war ursprünglich keine gestellte Anforderung.

Aufgrund der Tatsache, dass es allerdings Best-Fit und Worst-Fit gibt, wurde das Projekt um das Realisierungskonzept Last-Fit erweitert.

```
seekCounter=0 #Counter der die freien Plätze beinhaltet
for (( i=$(($save_speicherplatz-1)); i>=0; i-- )) #Schleife in der Größe des Speicherplatzes
do
---->if [[ -z "${save_speicher[$i]}" ]]; then #Wenn die i-te Stelle im Speicher leer ist
---->    seekCounter=$(($seekCounter+1)) #Aufeinander folgende leere Plätze werden hochgezählt
---->    if [[ $seekCounter -ge $seekGroesse ]]; then #Wenn der counter der leeren Plätze größer/gleich der unterzubnrGröße
---><------>index=$(($i+$seekGroesse-1))
---><------>LFCounter=1
---><------>for (( y=$index; $LFCounter<=$seekGroesse; y--)) #Schleife von hinten durchlaufen
---><------>do
---><------>    save_speicher[$y]=$seekGroesse
---><------>    LFCounter=$((LFCounter+1))
---><------>done
---><------>save_groesse[$index]=$seekGroesse #Größe an der Stelle im Array einfügen
---><------>save_name[$index]=$seekName #Name an der Stelle im Array einfügen
---><------>break
---->    fi
---->else
---->    seekCounter=0
---->fi
done
```

Abbildung 8. Logik eines Ausschnittes von dem Algorithmus für das Last-Fit Verfahren

Best-Fit und Worst-Fit

Best-Fit (siehe Abbildung 9, 10, 11 und 13) hat die Aufgabe den Platz im Speicher zu finden, wodurch es beim erfolgreichen Speichern möglichst kleine, wenn nicht sogar gar keine, Speicherfragmente hinterlässt.

Im Gegensatz zu anderen Realisierungskonzepten reicht es bei Best-Fit und Worst-Fit (siehe Abbildung 9, 10, 12 und 14) nicht nur aus mit einer einzigen Schleife zu arbeiten, da diese Realisierungen um einiges komplexer sind und deutlich mehr Intelligenz benötigen als es andere Verfahren tun.

```
seekCounter=0 #Counter der die freien Plätze beinhaltet
bereicheStelle=0 #Index
unset potentialStart
unset potentialEnd

for (( i=0; i<$save_speicherplatz; i++ )) #Schleife in der Größe des Speicherplatzes
do
    if [[ -z "${save_speicher[$i]}" ]]; then #Wenn die i-te Stelle im Speicher leer ist
        seekCounter=$(($seekCounter+1)) #Aufeinander folgende leere Plätze werden hochgezählt
    else
        if [[ $seekCounter -gt 0 ]]; then
            potentialStart[$bereicheStelle]=$(($i-$seekCounter)) #Start - Wenn der jetzige Platz belegt ist, aber der Counter bereits erhöht wurde
            potentialEnd[$bereicheStelle]=$(($i-1)) #Ende - Wenn der jetzige Platz belegt ist, aber der Counter bereits erhöht wurde
            bereicheStelle=$(($bereicheStelle+1)) #Index
        fi
---->   seekCounter=0 #Counter zurücksetzen
    fi
done
```

Abbildung 9. Logik eines Ausschnittes von dem Algorithmus für das Best-Fit und Worst-Fit Verfahren

```
if [[ $seekCounter -gt 0 ]] && [[ $i -eq $save_speicherplatz ]]; then #Counter wurde nicht zurückgesetzt und der Index gleicht der Speichergröße
---->potentialStart[$bereicheStelle]=$(($i-$seekCounter)) #Berechnung Start
---->potentialEnd[$bereicheStelle]=$save_speicherplatz #Berechnung Ende
fi
```

Abbildung 10. Logik eines Ausschnittes von dem Algorithmus für das Best-Fit und Worst-Fit Verfahren

```
kleinsteDifferenz=$(($save_speicherplatz+1)) #Auf einen unmöglichen Wert setzen
for (( i=0; i<${#potentialStart[*]}; i++ ))
do
---->for (( y=${potentialEnd[$i]}; y>=${potentialStart[$i]}; y-- ))
---->do
---->    platzGroesse=$(($platzGroesse+1)) #Berechnung der verfügbaren Größe
---->done
---->if [[ $platzGroesse -ge $seekGroesse ]]; then
---->    differenz=$(($platzGroesse-seekGroesse)) #Differenz zwischen verfügbarer Größe und der zu speichernden Größe
---->    if [[ $differenz -lt $kleinsteDifferenz ]]; then #Beim Erstaufruf immer legitim, da vorher unmöglich
---><------>kleinsteDifferenz=$differenz #Abspeichern der Differenz
---><------>kleinsteDifferenzStelle=${potentialStart[$i]} #Start-Stelle notieren
---><------>platzGroesse=0 #Reset der verfügbaren Größe
---->    fi
---->fi
done
```

Abbildung 11. Logik eines Ausschnittes von dem Algorithmus für das Best-Fit Verfahren

```
groessteDifferenz=-1 #Unmöglicher Startwert
unset groessteDifferenzStelle
for (( i=0; i<${#potentialStart[*]}; i++ ))
do
    unset platzGroesse
    platzGroesse=$((${potentialEnd[$i]}-${potentialStart[$i]}+1)) #Berechnung der Größe des verfügbaren Speicherplatzes
    if [[ $platzGroesse -ge $seekGroesse ]]; then
        differenz=$(($platzGroesse-$seekGroesse)) #Differenz zwischen Speicherplatzgröße und der zu speichernden Größe
        if [[ $differenz -gt $groessteDifferenz ]]; then
            groessteDifferenz=$differenz #Größte Differenz
            groessteDifferenzStelle=${potentialStart[$i]} #Startindex
        fi
    fi
done
```

Abbildung 12. Logik eines Ausschnittes von dem Algorithmus für das Worst-Fit Verfahren

In den Abbildungen 11 und 12 erkennt man den Logik-Unterschied der beiden Verfahren Best-Fit und Worst-Fit.

Bei dem Best-Fit Verfahren soll die Differenz zwischen Speicherblock und dem zu speichernden Wert so klein wie möglich sein, während bei dem Worst-Fit Verfahren die Differenz so groß wie möglich sein muss.

Diese Prüfungen erfolgen anhand berechneter Werte inklusive anschließender Ausschlussverfahren mittels Anweisungen mit -lt (less-than) bzw. -gt (greater-than).

```
counter=0
for (( y=$kleinsteDifferenzStelle; $counter<$(($seekGroesse)); y++ ))
do
    save_speicher[$y]=$(($seekGroesse))
---->counter=$((counter+1))
done
save_groesse[$kleinsteDifferenzStelle]=$(($seekGroesse)) #Größe an der Stelle im Array einfügen
save_name[$kleinsteDifferenzStelle]=$seekName #Name an der Stelle im Array einfügen
counter=0
```

Abbildung 13. Logik eines Ausschnittes von dem Algorithmus für das Best-Fit Verfahren

```
saveAt=$(($groessteDifferenzStelle+($differenz/2))) #Index für die Mitte des freien Speicherplatzes
counter=0
for (( y=$saveAt; $counter<$seekGroesse; y++ ))
do
    save_speicher[$y]=$seekGroesse #Abspeichern der Größe
    counter=$(($counter+1))
done
save_groesse[$saveAt]=$seekGroesse #Größe an der Stelle im Array einfügen
save_name[$saveAt]=$seekName #Name an der Stelle im Array einfügen
counter=0
```

Abbildung 14. Logik eines Ausschnittes von dem Algorithmus für das Worst-Fit Verfahren

Die Darstellung der Speicherbelegung

Die visuelle Darstellung der Speicherbelegung des jeweiligen Realisierungskonzeptes, das durch den Simulator ausgeführt wird, ist in drei Bereiche unterteilt (siehe Abbildung 15). Es wird sowohl die Speicherbelegung als auch die erstellten Partitionen mit ihren jeweiligen Namen und ihren jeweiligen Größen aufgeführt.

```
echo ${speicherout[*]}
echo "++++++++++"
    echo "----------"

    echo "**********"
    echo "Gespeicherte Namen: ${nameout[*]}"
    echo "**********"

    echo "----------"
    echo "Gespeicherte Größen: ${groesseout[*]}"
    echo "----------"
```

Abbildung 15. Ausschnitt der visuellen Ausgabe der Speicherbelegung

Die informative Darstellung listet relevante Informationen über den belegten Speicher auf (siehe Abbildung 16), die aus den jeweiligen Benutzereingaben heraus ermittelt werden können. Dazu zählen die Anzahl der Belegungen, die Anzahl freier Blöcke sowie der jeweils größte und kleinste Block im Speicher.

```
echo ""
echo "++++++++++"
echo "Belegung: $vo_belegterblock von $vo_max"
echo "Anzahl belegte Blöcke: $vo_belegterblock"
echo "Anzahl freie Blöcke: $(($vo_max-$vo_belegterblock))"
echo "Größter Block: $vo_groessterblock"
echo "Kleinster Block: $vo_kleinsterblock"
echo "++++++++++"
```

Abbildung 16. Ausschnitt der Ausgabe der Informationen der Speicherbelegung

Fazit

Zusammenfassend zeigt die Entwicklung des Projektes, dass die Erstellung eines Simulators zur Darstellung verschiedener Realisierungskonzepte der dynamischen Partitionierung mehrere Aspekte und drei Teilbereiche umfasst, die miteinander über mehrere Schnittstellen verknüpft, gegenseitig zu einem funktionsfähigen Programm zusammengefügt sind. Ohne das jeweilige Gegenstück funktioniert weder die Simulationssteuerung, noch die jeweiligen Realisierungskonzepte oder auch die Darstellung der Speicherbelegung. Besonders die Erarbeitung geeigneter Schnittstellen und der gegenseitige Zugriff auf diese ist besonders relevant. Durch die Plattform GitHub ist die gemeinsame Entwicklung des Projektes zur dynamischen Partitionierung mit den dazugehörigen Teilen gegeben, da der Austausch der einzelnen Teile dadurch ermöglicht wird. Schlussendlich zeigt sich, dass die erlernten Kenntnisse zu Bash und während der Programmentwicklung kontinuierlich durchgeführte Analysen zu dem Programmablauf sowie die Verbindung der Projektschnittstellen maßgeblich dazu beigetragen haben, das Projekt fertigzustellen.

Literaturverzeichnis

1.) Github. 2022. https://github.com/
2.) Baun, Christian. Betriebssysteme und Rechnernetze (SS2022). Werkstück A der Portfolioprüfung. 2022. http://www.christianbaun.de/BSRN22/index.html
3.) Baun, Christian. BSRN_Paper_Vorlage. Vorlage für die Abgabe von Werkstück A im SS2022. 2022.
4.) Baun, Christian. bsrn_SS2022_portfoliopruefung_teil_1_alternative_2.pdf. Protfolioprüfung-Werkstück A-Alternative 2. 2022. http://www.christianbaun.de/BSRN22/Skript/bsrn_SS2022_portfoliopruefung_teil_1_alternative_2.pdf
5.) Wolf, Jürgern. Shell-Programmierung. Rheinwerk Computing. 2004. https://openbook.rheinwerk-verlag.de/shell_programmierung/
6.) Stackoverflow. 2022. https://stackoverflow.com/
7.) GitHub Repository Link der Gruppe mit dem Abgabebranch (Abgabe-Werkstück-A): https://github.com/steffenrottke/DynamischePartitionierung/tree/Abgabe-Werkst%C3%BCck-A