Studienprogramm Sachbearbeiter:in Digitale Forensik
Praktikum Digitale Forensik Grundlagen
Dozenten: Tim Wetterau | Leander Hossfeld
wetterau@hs-mittweida.de | hossfeld@hs-mittweida.de
Stand: 27.03.2023

HOCHSCHULE MITTWEIDA
University of Applied Sciences

Digitale Forensik

Teil 2 – Grundlagen Linux II

In diesem Praktikum lernen Sie einige weitere Grundlagen zur Verwendung von Linux insbesondere bei der reinen Verwendung in der Kommandozeile. Hierzu nutzen wir wieder die bereits eingerichtete virtuelle Umgebung mit der VM Linux-Mint-BKA. Wir gehen davon aus, dass Sie stets in der Lage sind sich über Optionen der einzelnen Kommandos zu belesen, um die gestellten Aufgaben den Anforderungen entsprechend zu lösen.

Gegenstand dieses Praktikums sind folgende Punkte:

- Einfache Arbeit mit Dateien
- Nutzung von Texteditoren
- Nutzen von Operatoren
- Umgang mit grep (Regex)
- Umgang mit Prozessen
- Grundlagen Programmstrukturen

Vorbereitung

Zur Durchführung des Praktikums ist vorausgesetzt, dass die durch die Praktikumsanleiter zur Verfügung gestellte virtuelle Maschine Linux-Mint-BKA oder eine andere funktionierende Linux-Distribution auf ihrem Rechner erfolgreich installiert ist. Des Weiteren sollten Sie die in Praktikum 1 dargestellten Befehle und Operationen verstehen und anwenden können.

Hinweise:

➔ Auch für dieses Praktikum gilt für fortgeschrittene Nutzer, dass Sie dieses Praktikum überspringen können
➔ Wir weisen außerdem darauf hin, dass wir Ihnen abschließend zu den vorgestellten Befehlen eine Befehlsliste mit den wichtigsten zur Verfügung stellen
  - Dennoch sollen Sie dazu angehalten werden stets die Manpages der Kommandos zu konsultieren
  - Die Befehlsliste wird die Möglichkeiten und Mächtigkeit einzelner Befehle nicht vollumfänglich darstellen
  - Arbeiten Sie diese im Zuge des Praktikums im Modul „Betriebssysteme“ weiter aus und vervollständigen Sie diese für sich im Selbststudium.

Nutzen von Operatoren

Eine deutliche Vereinfachung in der täglichen Arbeit bieten Operatoren zur Umleitung und Steuerung von Operationen. Der einfachste Operator ist der zur Umleitung der Standardausgabe stout mit „**>**“. Damit kann die Ausgabe in eine Datei umgeleitet werden. Achtung: Mit der Nutzung von „**>**“ wird der bestehende Inhalt der Datei überschrieben. Soll der umgeleitete Inhalt angehängen werden, nutzen wir den Operator „**>>**“. Ein Beispiel dafür ist:

```
$ cat test.txt > new.file       # wenn new.file existiert, wird deren Inhalt überschrieben
$ cat test.txt >> append.file   # Inhalt aus test.txt wird an append.file angehängt
```

Man kann ebenfalls die Fehlerausgabe in eine Datei umleiten. Das erfolgt mit dem Operator „**2>**“ (zweiten Kanal). Somit wird eine Fehlermeldung bei Misserfolg einer Operation in eine Fehlerdatei umgeleitet. Wenn Sie sich in Ihrem Home-Verzeichnis befinden, probieren Sie den folgenden Befehl aus:

```
$ cp test.txt folder/test.txt 2> error.log
```

Beschreiben Sie die Funktionsweise des Befehls. Welche Ausgabe erwarten Sie? Überprüfen Sie Ihre Erwartungen im Dateisystem? Warum können Sie das geschehene Verhalten beobachten? Beheben Sie den Fehler, sodass keine Fehlermeldung mehr ausgegeben wird!

- **Soll test.txt in den Ordner folder kopieren**
- **Dieser existiert aber nicht**
- **Damit wird die entstehende Fehlermeldung nicht im Terminal angezeigt, sondern in die Datei error.log umgeleitet**
- **Überprüfen per ls -l oder ll**
- **mkdir folder**

Zwei logische Operator ist „**&&**“ und „**||**“. Sie stellen das logische **UND** und **ODER** dar. Damit lassen sich mehrere Befehle verknüpfen. Verwenden wir „**&&**“ mit der Syntax:

```
$ <Befehl1> && <Befehl2>
```

dann wird **Befehl2** nur ausgeführt, wenn **Befehl1** erfolgreich bearbeitet werden konnte. Hingegen ermöglicht „**||**“, dass **Befehl2** nur dann ausgeführt wird, wenn **Befehl1** nicht fehlerfrei (erfolgreich) ausgeführt wurde.

Ein weiterer essezieller Operator ist die Pipe „**|**“. Mit ihr ist es möglich, eine Ausgabe direkt an einen anderen Befehl weiterzuleiten, anstatt ans Terminal. Damit kann **Befehl2** direkt das Ergebnis von **Befehl1** weiterverarbeiten, ohne dass eine Zwischenspeicherung stattfindet. Damit können Sie z.B. einen Befehl schreiben, der alle Namen von Paketen, für die Updates verfügbar sind, in eine Datei schreibt. Dieser könnte wie folgt aussehen:

```
$ sudo apt update | tee upgradable.pkg
```

Einfache Arbeit mit Dateien

Die einfachste Art zur Interaktion mit Dateien ist das Ausgeben des Inhaltes der Datei. Dafür gibt es verschiedene Befehle, die je nach Benutzervorlieben oder Anwendungszweck angewendet werden. Der einfachste ist der Befehl:

```
$ cat [-option] <Dateiname>
```

Damit wird der Inhalt einer Datei in den Standardausgang ausgegeben. Normalenfalls ist das die Kommandozeile selbst. Also geben wir schließlich den Inhalt einer Datei auf die Kommandozeile aus. *Geben Sie den Inhalt der Datei test.txt in Ihrem Home-Verzeichnis aus, sodass jede Zeile eine Zeilennummer bekommt und Absatzzeichen sichtbar werden.*

```
$ cat -ne test.txt
```

Eine Alternative zur statischen Ausgabe des Testes ist das Ansehen des Inhaltes mit einem interaktiven Viewer. Dazu bietet sich der Pager **more** an. Damit kann man seitenweise den Inhalt ausgeben und von Seite zu Seite blättern. Ist der auszugebene Inhalt größer als eine Bildschirmseite, wird automatisch in den interaktiven Modus gewechselt. Dieser kann mit der Steuerung in Tabelle 1 bedient werden. Die Syntax wird folgend beschrieben:

```
$ more <Dateiname>
```

Tabelle 1: Steuerung des Pagers more

Taste	Aktion
Leertaste	Eine Bildschirmseite weiter
B	Eine Bildschirmseite zurück
F	Zwei Bildschirmseiten weiter
Enter	Eine Zeile weiter
S	Zwei Zeilen weiter
=	Ausgabe der aktuellen Zeilennummer
/	Suchbegriff eingeben, mit Enter bestätigen
v	vim-Editor öffnen (später mehr dazu)

Eine Alternative zu **more** ist **less**. Der Aufruf ist der gleiche, wie für **more**. Nur bietet **less** Vorteile im Gegensatz zu **more**: Anzeigen der Datei, obwohl diese noch nicht vollständig in den Speicher geladen ist, einfacheres Navigieren in den Dateien und intuitivere Suchmöglichkeiten. Die Steuerung wird durch Tabelle 2 dargestellt.

Tabelle 2: Steuerung des Pagers less

Taste	Aktion
Pfeiltasten	Zeileweise weiter-/zurückblättern
E / Y	wie Pfeiltasten
Bild auf / ab	Seite vor oder zurück
F / B	wie Bild auf / ab
Leertaste	Seite vor
<Zahl> Z / W	angegebene Zahl von Seite vor / zurück
G	Anfang des Dokuments
Shift + G	Ende des Dokumentes
/	Suchbegriff eingeben (kann Regex sein)

Ebenso kann man nur die ersten oder letzten Zeilen einer Datei ausgeben lassen. Dazu bieten sich die Kommandos

```
$ head <Datei>
$ tail <Datei>
```

an. Durch die Option „**-n**“ kann darunter auch gesteuert werden, wie viele Zeilen der entsprechenden Datei ausgegeben werden sollen. Das kann dann sinnvoll sein, wenn man nicht genau, was man in eine Datei geschrieben hat und nicht die komplette Datei ausgeben möchte. *Geben Sie die ersten 20 Zeilen der Datei **names.dict** in Ihrem Homeverzeichnis aus.*

```
$ head -n 20 names.dict
```

Ebenso können per Kommandozeile einfache Textoperationen durchgeführt werden. Darunter zählen folgende Kommandos:

```
$ sort <Datei1> [<Datei2> <Datei3> …]        # sortieren des Inhaltes
$ wc <Datei1> [<Datei2> <Datei3>]            # word count (Zählen von Zeilen und Wörtern)
$ tee <Dateiname>                            # Schreiben von Inhalt in eine Datei
```

Nutzen wir das Kommando sort, so können wir den Inhalt mehrerer Dateien auf einmal sortiert ausgeben. Das lohnt sich, wenn wir mehrere Dateien in einem Zusammenhang haben und nicht jede einzeln ausgeben wollen.

Dabei kann man auch auch verschiedenen Kriterien sortieren, wie z.B. nach Wörterbuch, unabhängig von Groß-/Kleinschreibung oder nach Dateigröße.

Per **wc** können Zeilen, Wörter und Zeichen in einer Datei ausgegeben werden. Dabei erscheint eine Ausgabe nach dem folgenden Schema:

```
$ Zeilenanzahl Wörteranzahl Zeichenanzahl Dateiname
```

Wenn die Ausgaben einzeln erfolgen sollen, können auch entsprechende Flags angegeben werden. Damit könnte man z.B. Variablen belegen, welche in einem Skript verarbeitet werden.

Zuletzt der Befehl **tee**. Tee nimmt die Eingabe über den Standardeingang der Kommandozeile und schreibt die Eingabe in eine Datei. **Achtung:** Standardmäßig wird damit der Inhalt einer bereits bestehenden Datei überschrieben. Mit „**-a**“ kann die Eingabe auch angehangen werden (append). Folgende Syntax gilt (hier mit dem Pipe-Operator als Eingabe – später mehr):

```
$ head -n 5 test.txt | tee -a test2.txt
```

Im gezeigten Befehl geben wir die ersten 5 Zeilen der Datei test.txt aus und leiten diese per Pipe an die Standardeingabe von **tee** weiter. **tee** hängt diese wiederum an die Datei **test2.txt** an.

Ein weiterer Befehl ist **paste** (einfügen). Mit **paste** haben wir die Möglichkeit den Inhalt einer Datei zeilenweise an den Inhalt einer anderen Datei anzuhängen. Haben wir also eine Datei mit Namen und eine andere mit Wohnorten können wir diese mit paste in einer Ausgabe übersichtlicher zusammenfassen. Die Syntax folgt dem Schema:

```
$ paste [-option] <Datei1> <Datei2>
```

Wiederrum kann man mit cut verschiedene Spalten einer Datei einzeln ausgeben. Dazu müssen diese aber ein einheitliches Trennzeichen haben (**Leerzeichen, ,, ;, |, -| TAB,** etc.). Wir nutzen folgende Syntax:

```
$ cut [-d<Delimiterzeichen] [-f<Spaltenliste>] <Dateiname>
```

Dabei ist zu erwähnen, dass das Standardtrennzeichen für **paste** und für **cut TAB** ist.

Bearbeiten Sie dazu folgende Aufgaben:

1. *Schauen Sie sich die Dateien names.dict und schools.dict in den vorgestellten Editoren an.*
    a. *Machen Sie sich mit deren Anwendung vertraut*
2. *Geben Sie die letzten 10 Zeilen der Datei names.dict aus.*
3. *Wie viele Zeilen haben die beiden Dateien jeweils?*
4. *Fügen Sie zeilenweise die Dateien names.dict und schools.dict zusammen.*
    a. *Trennzeichen Tabulator*
    b. *Schreiben der Ausgabe in eine neue Datei concat.dict*
5. *Sortieren die entstandene Liste nach Namen.*
6. *Geben Sie die Nachnamen und Orte aus concat.dict aus.*
7. ***Zusatz:** Geben Sie erneut Nachnamen und Orte aus sortieren Sie diese nach den Namen.*
    a. *Schreiben Sie das Ergebnis in eine neue Datei sortedConcat.dict*

```
$ more / less names.dict / schools.dict
$ tail -n 10 names.dict
$ wc -l names.dict / schools.dict
$ paste names.dict schools.dict | tee concat.dict
```

```
$ sort concat.dict
$ cut -f2,3 concat.dict
$ Zusatz:  cut -f2,3 concat.dict | tee sortedConcat.dict
```

Nutzung von Texteditoren

Neben dem Schreiben in Dateien per tee oder Pipes „|“, können Dateien auch direkt editiert werden. Die beiden bekannsten Editoren für die Kommandozeile sind vi und nano. Deutlich intuitiver und leichter zu bedienen ist nano. Hingegen gibt es für vi die Weiterentwicklung vim, welche deutlich anpassungsfähiger ist. vi wird eher weniger verwendet und bedarf einer hohen Einarbeitung. An der Stelle sei darauf verwiesen, dass die Arbeit mit vi zwar möglich ist, aber meist nicht nötig. Sollten Sie dennoch Interesse an dessen Funktionsweise haben verweisen wir auf die Interquelle der TU Chemnitz verwiesen.

Wir wollen uns hier auf den Editor nano beschränken. Rufen wir den Editor auf legt dieser eine neue Datei an, wenn die angegebene Datei nicht existiert, anderfalls öffnet er nur die angegebene Datei. Rufen Sie den Editor über den folgenden Befehl auf:

```
$ nano new.file
```

Daraufhin öffnet sich der Editor mit einer neuen Datei und wir können fröhlich drauf los editieren. Grundlegend ist die Steuerung intuitiv und sie können sich mit den Pfeiltasten als auch mit den Bildlauftasten im Editor bewegen. Im unteren Teil sehen Steuerungshinweise. Grundlegende Hilfe erhalten Sie außerdem über die Tastenkombination Strg + G. Schreiben Sie ein paar Zeilen in den Editor.

Entgegen der Nutzung von Windows, können wir die geänderten Daten nicht mit Strg + S speichern. Dazu nutzen wir Strg + O (write out). Einige weitere Tastenkombinatinen finden Sie in Tabelle 3.

Tabelle 3: Tastenkombination zur Bedienung von nano

Tastenkombi	Funktion
Strg + O	Speichern
Strg + X	Beenden
Strg + W	Text suchen
Alt + A	Anfangsmarkierung setzen (mit Pfeiltasten weiter markieren)
Alt + 6	Kopieren
Strg + K	Ausschneiden
Strg + U	Einfügen
**Strg + **	Ersetzen

Ebenso bietet sich nano standardmäßig dazu an, Bash-Skripte zu schreiben. Es bietet ein entsprechendes Highlighting der Syntax. Bash-Skripte werden allgemein immer in der ersten Zeile begonnen mit:

```
$ #!/bin/bash
```

Damit weiß das Betriebssystem, dass diese Datei mit der Bourne-Again Shell (bash) interpretiert werden müssen. Dazu aber später mehr.

Umgang mit grep (Regex)

Reguläre Ausdrücke (regex – regular expressions) sind ein mächtiges Tool zur Suche in Texten. Die Besonderheit bei regulären Ausdrücken ist, dass wir keinen festen String vorgeben, sondern nur ein Muster, auf welches dann verschiedene Treffer matchen. Ein alltägliches Beispiel soll uns der Name Meier dienen, in all seinen

Schreibformen mit ei, ai, ey, oder ay. Wollen wir alle Personen finden, die einen dieser Namen tragen, müssten wir nach allen Schreibformen suchen. Regex erspart uns diese Arbeit. Mit dem folgenden Ausdruck können wir alle Schreibweisen auf einmal finden:

```
$ grep M[ae][iy]er [<Dateiname>]
```

Legen Sie sich eine Datei neue Datei mit nano an und schreiben Sie in diese verschiedene Namen, welche ähnliche Schreibweisen haben. Nehmen Sie gern das Beispiel Meier und testen sie den oben geschrieben Befehl in der Kommandozeile. Wie lautet der gesamte Befehl? Wie kann der Befehl unter Verwendung eines Pipe-Operators aussehen?

```
$ Grep M[ae][iy]er datei
$ Cat datei | grep M[ae][iy]er
```

Um bestimmte Muster anzugeben, brauchen wir Bausteine, die es uns ermöglichen verschiedene Schreibweisen zu erfassen. Wir unterscheiden Symbole, Charakterklassen und Quantifier. Symbole sind einzelne Zeichen, die eine bestimmte Funktion erfüllen in Bezug auf den gestellten Ausdruck. Charakterklassen stellen Gruppierungen von Zeichen dar, aus welchen der Ausdruck sozusagen „auswählen“ kann, z.B. Kleinbuchstaben. Quantifier realisieren das Zählen des Vorkommens eines Zeichens oder einer Charakterklasse. Als Referenz und zur Hilfe haben wir Ihnen ein Dokument auf Moodle zur Verfügung gestellt, welches die wichtigsten Elemente von Regex umfasst.

Lösen Sie anhand der Referenz folgende Aufgaben:

- Nennen Sie jeweils drei Beispiele, welche auf die folgenden Regulären Ausdrücke passen:
  - ^[(w{3})(http[s]?)]\..*.de
  - [:digit:,]*[€\$]
  - .*@g(oogle)?mail\.[(com)(de)]$
- Schreiben sie selbstständig Regex, welche folgende Wortgruppen erfassen:
  - Doppelnamen mit Bindestrich
  - Telefonnummern (mit und ohne Ländervorwahl)
  - IBANs
  - Zusatz: E-Mail Adressen unabhängig vom Provider

```
$ Doppelnamen:          [[:alpha:]]*-[[:alpha:]]*
$ Telefonnumern:        [0+]?[[:digit:] ]{8,15}
$ IBAN:                 [[:upper:]]{2}[[:digit:]]{2}[[:digit:] ]{18,23}
$ E-Mail Adressen:      [[:alnum:][:punct:]]*@[[:alnum:][:punct:]]*\.[[:lower:]]{2,3}
```

Gegebenenfalls müssen Sie für die Befehle zusätzlich die Option „-E“ verwenden, da Quantifier, wie {n,m} nicht in der grundlegenden, sondern in der erweiterten Syntax vorkommen können. Reguläre Ausdrücke sind für verschiedene Anwendungen gut, wie sie schon in den Aufgaben gesehen, können wir auch Telefonnumern damit herausbekommen. Forensische Suiten, wie z.B. XWays Forensics, erlauben es auch Asservate mit regulären Ausdrücken in parallelen Suchen zu analysieren. Es folgen Beispiele bei denen Regex Anwendung finden können:

- Suche nach nach Gehör geschriebenen Wörtern
- Suche nach Loginformationen
- Suche nach BitLocker-Schlüsseln
- Überprüfen, ob ein Text in Eingabefeldern valide ist

Grundlagen Programmstrukturen

Als letzten Punkt in dieser Praktikumssitzung wollen wir über Programmstrukturen sprechen. Sie sind essenziell zur bedingungsgesteuerten Automatisierung von repetitiven Arbeitsschritten. Die Programmstrukturen im Linux Bash-Skripting gleichen rein strukturell denen anderer Programmiersprachen, wie Java oder C. Das Kapitel ist vorbereitend für das letzte Praktikum, in dem Sie selbst mit drei einfachen Skripten konfrontiert werden.

Variablen

Variablen sind wichtig zur Zwischenspeicherung von Werten, die verarbeitet werden sollen und mehrmals verwendet werden. Dazu nutzen wir die Syntax:

```
$ <Variablenname>=<Zahl>
$ <Variablenname>='<String>'
```

Damit haben wir den Variablennamen mit einem Wert belegt. Wollen wir diesen Wert wieder abrufen, nutzen wir das **$**-Zeichen und den Namen der Variable, wie in folgender Syntax:

```
$ $<Variablenname>
$ echo $<Variablenname>  # gibt den Wert auf der Kommandozeile aus
```

Ebenso können Variablen das Ergebnis von anderen Befehlen zugewiesen bekommen. Dazu nutzen zu dem **$** noch runde Klammern, wie in folgendem Beispiel:

```
$ datum=$(date)   # zuweisen des Datums zur Variable datum
$ echo $datum     # ausgaben des Wertes von datum auf derm Terminal
```

Für die Berechnung von Zahlen verwenden wir den Befehl **expr** mit der folgenden Syntax:

```
$ expr <Berechnung>
$ expr 5 + 5                    # Beispielrechnung
$ 10
```

*Berechnen Sie wie viele Tage ein Monat haben müsste unter der Annahme, dass ein Jahr aus **365** Tagen und **12** Monaten besteht. Speichern Sie das Ergebnis in der Variable month. Schreiben Sie einen **einzeiligen Befehl**, der diese Aufgabe erfüllt. **Zusatz:** Schreiben Sie die Variable noch in eine Datei month.txt. Beide Befehle sollen zusammen auf eine Zeile geschrieben werden. Nutzen Sie Operatoren.*

```
$ month=$(expr 365 / 12)
$ month=$(expr 365 / 12) && echo $month > month.txt
```

Verzweigungen

Verzweigungen dienen dazu einen bedingungsgesteuerten Progammablauf zu realisieren. Z.B Zur Reaktion auf Benutzereingaben oder die Übergabe von Flags an das Skript selbst. Wir prüfen dazu grundlegend immer einen Sachverhalt gegenüber einer Bedingung, die der Sachverhalt erfüllen muss. Also wenn x, dann y, sonst z. Die einfachste Möglichkeit dazu bietet die IF-Verzweigung. Sie unterliegt in der bash der folgenden Syntax:

```
if [ <Bedinungsprüfung1> ]
then
      <Ausführungsblock1>
elif [ <Bedinungsprüfung2> ]
```

```
then
      <Ausführungsblock2>
else
      <ELSE-Ausführungsblock>
fi
```

Die Syntax ist so zu verstehen, dass wir in der Klammer nach IF eine Prüfung auf die gegebene Bedingung durchführen. Wird diese erfüllt, gehen wir in den Ausführungsblock1. Das folgende ELIF bedeutet „else if“, also übersetzt „ansonsten wenn“. Es wird nur dann geprüft, wenn die Bedinungsprüfung1 nicht erfüllt wurde. Wenn auch die Bedingungsprüfung2 nicht erfüllt wurde, dann wird unweigerlich der ELSE-Block ausgeführt. In ELSE stehen Bedingungen, die ausgeführt werden, wenn keine der vorherigen Bedingungsprüfungen erfüllt wird. Abgeschlossen wird der IF-Block syntaktisch mit dem Wörtchen FI also einem IF rückwärts. Die Anzahl der IF-Blöcke lässt sich beliebig erweitern und ELSE-Blöcke sind nicht zwingend erforderlich, was aber je nach Anwendungsfall vom Programmierer entschieden werden muss.

Für die Bedingungsprüfungen gibt es Bedingungsausdücke, die sich auch nach Anwendungsfall unterscheiden, z.B. ob wir auf Zahlen, Wörter oder Dateisystempfade prüfen. Einige wichtige sind in Tabelle 4 aufgezeigt.

Tabelle 4: Bedingungsausdrücke für das Bash-Skripting

Ausdruck	Beschreibung	Eselsbrücke
Pfade		
`-d`	Angegebener Pfad ist ein Verzeichnis	directory
`-f`	Angegebener Pfad ist eine Datei	file
`-r`	Angegebener Pfad ist lesbar	readable
`-w`	Angegebener Pfad ist schreibbar	writable
Strings		
`-z`	Länge des Strings ist Null	zero
`-n`	Länge des Strings ist > 0	not zero
`==`	String1 ist gleich String2	
`!=`	String1 ist ungleich String2	
Mathematische Ausdrücke		
`-eq`	Zahl1 ist gleich Zahl2	equals
`-ne`	Zahl1 ist ungleich Zahl2	not equals
`-gt`	Zahl1 ist größer als Zahl2	greater than
`-ge`	Zahl1 ist größer gleich Zahl2	greater equals
`-lt`	Zahl1 ist kleiner als Zahl2	less than
`-le`	Zahl1 ist kleiner gleich Zahl2	less equals

Beispiel: Wir wollen ein Skript schreiben, welches prüft ob Freitag ist und eine entsprechende Zeile ausgibt 😊

```
#!/bin/bash                                          # Erklärung, dass Skript durch bash interpretiert wird

tag=$(date +“%A“)                                    # Zuweisen von tag, +“%A“ gibt Wochentag zurück
if [ $tag == „Freitag“ ]                             # Prüfen, ob tag gleich „Freitag“ ist
then                                                 # wenn ja, dann
      echo „Juhu es ist Freitag, Morgen ist Wochenende“      # Ausgeben dieses Textes
elif [ $tag == „Samstag“ || $tag == „Sonntag“ ]      # sonst, wenn tag gleich Samstag ODER Sonntag
```

```
then                                                              # dann
      echo „Es ist schon Wochenende“                              # diesen Text ausgeben
else                                                              # wenn nichts davon erfüllt, dann
      echo „Es ist noch kein Wochenende in sicht. Arbeite weiter …“   # diesen Text ausgeben
fi                                                                # Ende If-Verzweigung
```

Lesen Sie sich die Kommentare, also alles, was hinter den „#“ steht, durch und versuchen Sie zu verstehen, wie das Skript arbeitet. Übernehmen Sie das Skript in eine neue Datei mit nano. Sie können dazu die gemeinsame Zwischenablage von Gast- und Hostsystem verwenden. Einfach kopieren und in nano mit der Tastenkombination Strg + Shift + V einfügen. Speichern Sie die Datei und verlassen Sie nano wieder. Machen Sie das Skript ausführbar (Zugriffsrechte ändern) und führen Sie es anschließend mit dem folgenden Befehl aus und schauen Sie, was auf dem Terminal ausgegeben wird.

```
$ ./<Skriptname>
```

```
$ Chmod u+x friday.sh
```

Alternativ können Verzweigungen auch mit einer case-Verzweigung realsiert werden. Das geht aber an der Stelle zu tief in die Programmierung hinein und wird daher nicht weiter beleuchtet. Belesen Sie sich bitte selbst über dessen Anwendung bei Interesse. Dazu sei Ihnen folgende Quelle gegeben:

- Bash – Arbeiten und programmieren mit der Shell – opensuse.org

Schleifen

Ein weiteres Element der bedingungsgesteuerten Programmierung ist die Schleife. Also das Wiederholen einer Aktion bis eine Bedingung erfüllt oder nicht mehr erfüllt, je nach Anwendung. Dabei unterscheiden wir zwischen folgenden Schleifenarten in der Bash:

- while
- until
- for

Wir bitten Sie sich aufgrund des beschränkten Rahmens mit den while und until-Schleifen selbst in der gegeben Quelle zu beschäftigen. Die for-Schleife soll aber näher beschrieben werden, da Sie diese brauchen, um das Praktikum 4 zu bearbeiten. Sie ist dazu da eine Folge von Befehlen mit einer vorher bekannten Anzahl zu wiederholen. Das kann sinnvoll sein, aber je nach Anwendungsfall auch nicht. Sehen wir uns die Syntax der for-Schleife an:

```
For <zähler> in {<min>..<max>}          # <min> und <max> sind natürliche Zahlen.
Do
      <Ausführungsblock>
Done
```

Achten Sie beim Setzen des Rahmens in den geschwungenen Klammern auf die ZWEI Punkte dazwischen ohne Leerzeichen. Im Ausführungsblock können wieder beliebige Befehle ausgeführt werden, welche nacheinander ausgeführt werden und wiederholt werden sollen.

Beispiel: Wir wollen ein Skript, welches im Terminal von 0 bis 10 zählt.

```
#!/bin/bash

for i in {0..10}
do
```

```
        echo $i
done
```

Die Grenzen sind uns vorher bekannt und können daher problemlos in den geschwungenen Klammern gesetzt werden. Es ist zu beachten, dass wir i als unsere Zählvariable nutzen. Bei jedem Durchlauf der Schleife wird i AUTOMATISCH inkrementiert, also um 1 erhöht. Und per echo geben wir den aktuellen Zustand von i aus.

Übernehmen Sie auch dieses Skript in Ihre VM und testen Sie. Es steht Ihnen frei, die beiden Skripts auch ein wenig zu manipulieren und zu schauen, was Ihre Änderungen bewirken. Probieren ist besser als Studieren, fürchten Sie sich nicht vor Fehlern.