Portfolioprüfung im Kurs SS2020 „Betriebssysteme und Rechnernetze: Werkstück A – Alternative 4 Thema: Scheduling-Verfahren

Fabio Sude
Fabio_Sude@gmx.de

Rashan Can Mitchell
rashan_cm@icloud.com

Frankfurt, 08.Juli 2020

Zusammenfassung: Dieses Dokument beschreibt die Vorgehensweise bei der Erstellung eines Programms zur Simulation der Scheduling-Verfahren „First Come First Served“ und „Round Robin“ mit einem frei definierbarem Zeitquantum. Für die Programmierung wurde die Skript-Sprache Bash verwendet.

1 Einleitung

Scheduling-Verfahren sind ein wesentlicher Bestandteil aller Betriebssysteme. Sie ermöglichen es entweder eine Vielzahl von Anwendungen parallel zu betreiben und dabei die System-Ressourcen so effektiv wie möglich auszunutzen, Routineaufgaben zu automatisieren oder innerhalb fest definierter Zeitintervalle ein Ergebnis zu liefern. Je nach Anwendungsgebiet kommen in der Programmierung unterschiedliche Scheduling-Verfahren zum Einsatz. Diese sind ist abhängig von der gewählten Programmiersprache und dem verwendeten Betriebssystem.

Aus diesem Grund bietet sich für die Simulation der Scheduling-Verfahren die Programmierung als Bash-Shell-Skript an, da hier ein plattform- und betriebssystemunabhängiges Programm erstellt werden kann.

Für die passende Auswahl des Scheduling-Verfahrens muss der Entwickler entscheiden ob die Anwendung im Batch- oder im Dialogbetrieb laufen soll. Für den Batchbetrieb, auch Stapelverarbeitung genannt, müssen alle Daten zum Programmstart vorliegen. Die Batch-Programme werden dann in die Queue gestellt und sequentiell abgearbeitet, eine Interaktion mit dem Anwender ist nicht notwendig.

Bei Dialoganwendungen kommt es darauf an, dass das Programm unterbrochen werden kann, Daten aus anderen Programmen übernimmt oder Teilaufgaben an weitere Prozesse oder Programme weitergibt und auf Rückmeldung wartet.

Aus Sicht des Anwendungsentwicklers ist es daher das Verständnis für die unterschiedlichen Scheduling-Verfahren, deren Einsatzgebiete und Vor- & Nachteile für die effektive Erstellung von Programmen essentiell.

Mit dem programmierten Simulator werden zwei unterschiedliche Scheduling-Verfahren simuliert. Für das „Werkstück A – Alternative 4“ war ursprünglich die Simulation von drei der Scheduling-Verfahren vorgesehen. Es wurde jedoch auf zwei unterschiedliche Verfahren reduziert und diese sind in einem modular aufgebauten Programm abgebildet. Das Programm ist um beliebig viele simulierte Scheduling-Verfahren erweiterbar.

Der Ansatz in der Erstellung des Programms so ist so konzipiert, dass Team getrennt an den einzelnen Programmodulen arbeiten kann und die Arbeitsergebnisse dann in einem Programm zusammengeführt werden.

2 Vorstellung der Arbeit

Das in dieser Dokumentation beschriebene Programm simuliert die Scheduling-Verfahren „First Come First Served“, im folgenden auch FIFO genannt und „Round Robin“, welche als Bash-Skript implementieret worden sind.

Bei der Erstellung des Scripts wurde darauf Wert gelegt ein modulares Programm zu entwickeln, das die Teamarbeit begünstigt, um zusätzliche Simulationen erweiterbar ist und den Anwendungsentwicklungsprozess durch eine Debugging-Funktion transparenter macht.

Die Menüstruktur und Bedienung des Programms ist möglichst einfach konzipiert, verfügt aber über einfach Fehlerbehandlungsroutinen. Auf eine Hilfefunktion und grafische Darstellung wurde verzichtet.

Hauptaugenmerk liegt auf der strukturierten Programmierung in der Skript-Sprache Bash und der Wiederverwendbarkeit einzelner Code-Elemente.

3 Vorgehensweise bei der Implementierung

Im ersten Schritt wurde sich mit der Architektur der Anwendung beschäftigt und das Programm in einzelne Module aufgeteilt:

1. Hauptprogramm: ruft die einzelnen Module auf und bietet einen generische Benutzerdialog zur Auswahl der einzelnen Simulationen.
2. Programm-Modul A: simuliert das FIFO-Verfahren und stellt die Ergebnisse auf dem Bildschirm dar, ist auch allein lauffähig
3. Programm-Modul B: simuliert das Round-Robin-Verfahren und stellt die Ergebnisse auf dem Bildschirm dar, ist auch allein lauffähig

Der Souce-Code wurde zur Team-Arbeit auf GitHub abgelegt und steht dort zur Überprüfung zur Verfügung:

https://github.com/FabioSude/Portfoliopruefung-Werkstueck-A-BsRn

Im nächsten Schritt wurde sich mit den Möglichkeiten der Programmierung in de Bash beschäftigt, um die Aufgabenstellung der Simulation von Scheduling-Verfahren möglichst effektiv zu erledigen. Dazu haben wir neben den zur Verfügung gestellten Lehrmaterial zur Linux-Systemadministration und Shell-Programmierung zahlreiche Quellen im Internet konsultiert.

Dies ist anhand der, vom Dozenten, vorgelegten Skripte, sowie andere Plattformen wie beispielweiße YouTube realisiert worden. Um die ersten Implementierungen jedoch umsetzen zu können, muss erst ein Grundverständnis über die verschiedenen Scheduling-Verfahren geschaffen werden. Die erste Überlegung hierbei galt dem Ablauf der verschiedenen Schritte, der einzelnen Algorithmen. Da das FIFO Scheduling-Verfahren als Grundlage für das Round Robin Verfahren dient, zahlt es sich als sinnvoll aus, sich mit diesem als erstes zu beschäftigen. Das lässt sich daran festmachen, dass sich die Prozesse in einer zyklischen Warteschlange nach dem FIFO Prinzip einreihen. Sollte das Zeitquantum „unendlich“ sein, verhält sich das Round Robin wie das FIFO-Verfahren.

3.1 Entwicklung der Anwendung

Das „First Come First Served“-Verfahren (im folgenden FIFO genannt), so kann man schon anhand des Namens ableiten, dass bei diesem Verfahren die Prozesse, in der Reihenfolge der CPU zugewiesen werden, wie sie gestartet wurden. Die Verarbeitung erfolgt streng sequenziell.

Beim Round-Robin-Verfahren werden für jeden Prozess Zeitscheiben von einer bestimmten Dauer festgelegt, das sogenannte Zeitquantum. Alle Prozesse werden in einer zyklischen Warteschlange nach dem FIFO-Prinzip eingereiht und jeder Prozess bekommt solange Zugriff auf die CPU, wie durch das Zeit-quantum festgelegt wurde. Nach Ablauf des Zeitquantums wird dem Prozess der Zugriff auf die CPU entzogen und der Prozess kommt an das Ende der Warteschlange. Dieses Rundlauf-

Verfahren wird solange wiederholt, bis der Prozess abgearbeitet ist. Nach Beendigung wird dieser aus der Warteschlange entfernt.

Das Ziel der Simulation der beiden Verfahren ist eine dynamische Eingabe der Anzahl der Prozesse, deren jeweilige Laufzeiten, sowie Ankunftszeiten. Um dies zu realisieren ist es von Vorteil, das Programm als erstes statisch zu entwickeln und dieses darauffolgend Schritt für Schritt zu einem dynamischen Programm zu erweitern. Um ein statisches Programm zu entwickeln, sind die Laufzeiten, die Ankunftszeiten und die Anzahl der Prozesse zu initialisieren. Als nächstes gilt es den Algorithmus zur Berechnung der Laufzeiten mit einfließen zu lassen.

```
43 echo -e "\nLaufzeiten der Prozesse: "
44 # Berechnung Wartezeit
45 waitingTIME[1]=0
46 # Berechnung Laufzeit
47 rtimeNEU[1]=$(( ${rtime[1]}-${atime[1]} ))
48
49 waitingTIME[2]=$(( ${rtime[1]}-${atime[2]} ))
50 rtimeNEU[2]=$(( ${rtime[1]}+${rtime[2]}-${atime[2]} ))
51
52 waitingTIME[3]=$(( ${rtime[1]}+${rtime[2]}-${atime[3]} ))
53 rtimeNEU[3]=$(( ${rtime[1]}+${rtime[2]}+${rtime[3]}-${atime[3]} ))
54
55 waitingTIME[4]=$(( ${rtime[1]}+${rtime[2]}+${rtime[3]}-${atime[4]} ))
56 rtimeNEU[4]=$(( ${rtime[1]}+${rtime[2]}+${rtime[3]}+${rtime[4]}-${atime[4]} ))
57
58 averageRUNTIME=$(( ${rtimeNEU[1]}+${rtimeNEU[2]}+${rtimeNEU[3]}+${rtimeNEU[4]} ))
59
60 averageRUNTIMEneu=$(( ${averageRUNTIME} / 4 ))
61
62 averageWAITINGTIME=$(( ${waitingTIME[1]}+${waitingTIME[2]}+${waitingTIME[3]}+${waitingTIME[4]} ))
63
64 averageWAITINGTIMEneu=$(( ${averageWAITINGTIME} / 4 ))
65
66 builtin unset averageRUNTIME averageWAITINGTIME
```

Abbildung 1: Ausschnitt aus FIFO-Verfahren statisch

Wie man anhand des in **Abbildung 1** gezeigten Code-Ausschnittes sehen kann, erhält man nach dessen Ausführung die Laufzeiten und Wartezeiten der einzelnen Prozesse. Jedoch ist die Berechnung dieser Zeiten noch nicht elegant gelöst. Es handelt es sich um eine triviale Lösung, der zuvor vorgestellten Herausforderung, welche dennoch zur einer Ergebnisführung beiträgt. Nach dem Entwickeln einer ersten erfolgreichen Lösung des Algorithmus, gilt es als nächstes, diesen zu erweitern und zur Vollständigkeit dynamisch zu gestalten. Dazu gehört nicht nur die korrekte Ausgabe der Werte, sondern auch die Gestaltung des Codes, sowie dessen Syntax.
Eine weitere Problematik bei diesem Thema, stellt die mögliche Falscheingabe der Prozesslaufzeiten dar.
Um das zu erreichen ist ein IF-Statement zu implementieren, wie in **Abbildung 2** zu sehen ist, welches den eingegeben Variablen-Wert mit sich selbst vergleicht, um zu überprüfen, ob es sich bei der Eingabe um einen Integer handelt (Z. 30). Ist dies nicht der Fall, so kommt es zur Löschung der Eingabe aus dem Array (Z. 41) und damit zu einer erneuten Eingabeaufforderung der CPU-Laufzeit.

```
30 while [ ${#rtime[@]} -lt ${anzR} ]
31 do
32         read -p "Prozess ${anzahlCOUNT}: " rtime[${anzahlCOUNT}]
33
34         # Vergleicht Variable mit sich selbst, um zu schasuen ob sie ein Integer ist, ansonsten wird Eingabeaufforderung wiederholt
35         # Fehlermeldung wird nicht auf dem Display angezeigt, ein auszugebender Datenstrom wird verworfen -> "&>/dev/null"
36         if test "${rtime[${anzahlCOUNT}]}" -eq "${rtime[${anzahlCOUNT}]}" &>/dev/null
37         then
38                 anzahlCOUNT=$(( ${anzahlCOUNT} + 1 ))
39         else
40                 # "builtin unset rtime[${anzahlCOUNT}]" -> Fehlerhafte Eingabe wird aus dem Array gelöscht
41                 builtin unset rtime[${anzahlCOUNT}]
42         fi
43 done
```

Abbildung 2: Abfangen von falschen Eingaben

Der nächste Schritt, wie in **Abbildung 2** zu sehen, besteht in einer dynamischen Implementierung des FIFO-Verfahrens (Z. 28). Dort ist zusehen, dass die Anzahl der Prozesse nicht mehr im Code selbst vorgegeben, sondern durch die Variable „anzR“ vom Benutzer selbst zu belegen ist. Gleiches wird dann für die Ankunftszeit wiederholt. Darauffolgend muss nun der zuvor geschriebene Code zur Berechnung der Lauf- und Wartezeiten sich zu einem dynamischen Algorithmus weiterentwickeln. Dies geschieht durch das Einsetzen von Schleifen. Durch eine FOR-Schleife kann jedes einzelne Element im Array angesteuert und für die Berechnung verwendet werden.

```
 98 while [ ${numberPROCESS} -ne ${#rtime[@]} ]
 99 do
100         # Es wird jeder Index im Array durchlaufen
101         for index in ${!openRUNS[@]}
102         do
103                 # Laufzeiten berechnen
104                 # In "runTIME" wird die Laufzeit aus "${rtime[${index}]}" gespeichert
105                 runTIME=$(( ${runTIME} + ${rtime[${index}]} ))
106                 # Es wird ein Array "runtimeFINAL" ertsellt, dort wird die Laufzeit des Prozesses berechnet
107                 runtimeFINAL[${index}]=$(( ${runTIME} - ${atime[${index}]} ))
108                 # In averageRUN werden die Laufzeiten aufaddiert, um später den Durchschnitt zu berechnen
109                 averageRUN=$(( ${averageRUN} + ${runtimeFINAL[${index}]} ))
110
111                 # Wartezeit berechnen
112                 # Laufzeit des Prozesses wird minus die CPU-Laufzeit gerechnet -> Wartezeit
113                 waitingTIME=$(( ${runtimeFINAL[${index}]} - ${rtime[${index}]} ))
114                 # Es wird ein Array "waittingtimeFINAL" erstellt, dort werden die Wartezeiten der Prozesse berechnet
115                 waitingtimeFINAL[${index}]=$(( ${waitingtimeFINAL[${index}]} + ${waitingTIME} ))
116                 # In "averageWAIT" werden die Wartezeiten aufaddiert, um später den Durchschnitt zu berechnen
117                 averageWAIT=$(( ${waittingtimeFINAL[${index}]} + ${averageWAIT} ))
118
119                 if [ "${debug}" = "yes" ]
120                 then
121                         echo -e "DEBUG: Laufzeit  fuer Prozess ${index}: ${runtimeFINAL[${index}]} ms"
122                         sleep 0.2
123                         echo -e "DEBUG: Wartezeit fuer Prozess ${index}: ${waittingtimeFINAL[${index}]} ms\n"
124                         sleep 0.2
125                 fi
126
127                 # counter, identisch zu let numberPROCESS=numberPROCESS + 1
128                 numberPROCESS=$(( ${numberPROCESS} + 1 ))
129         done
130
131 done
132 builtin unset runTIME numberPROCESS
```

Abbildung 3: Dynamischer Algorithmus zu Berechnung der Laufzeiten (FIFO)

Wie in **Abbildung 3** zu sehen, ist in den Zeilen 98 – 109 der eigenentwickelte Algorithmus für die Berechnung der Laufzeiten der Prozesse zu sehen. In der Variablen „runTIME“ (Z. 105) erfolgt die Speicherung der Laufzeiten. Diese werden in jedem Durchlauf aufeinander addiert, wobei der Wert zu Beginn 0 beträgt. Dann wird ein Array namens „runtimeFINAL“ (Z. 107) deklariert. Dort ist das Ergebnis aus der Rechnung Laufzeit minus Ankunftszeit gespeichert. Dieses Ergebnis ist die Laufzeit, die der Prozess benötigt, um von der CPU verarbeitet zu werden. Diese Laufzeit wird

dann in der Variable „averageRUN“ (Z. 109) gespeichert, welche zu Beginn ebenfalls mit dem Wert 0 initialisiert worden ist.
In den Zeilen 113 – 117 werden die Wartezeiten berechnet, welche auf dem vorherigem Rechenschema basieren. Im Gegenzug zu diesem, wird hier die Gesamtlaufzeit des Prozesses von der Laufzeit der CPU subtrahiert. Dieser Wert wird in der Variablen „waitingTIME“ (Z. 113) gespeichert, welche den Wert 0 zu Anfang hat. Jetzt wird ein Array „waitingtimeFINAL“ (Z. 115) erstellt. Dort wird dann an der ersten Stelle des Arrays „waitingtimeFINAL“ der Wert aus der Variable „waitingTIME“ draufaddiert. Der an der ersten Stelle befindliche Wert, wird mit der Variable „averageWAIT“ (Z. 117), welche ebenfalls mit 0 initialisiert wurde, addiert. Diese Variable ist nötig für die spätere Berechnung der durchschnittlichen Wartezeit. Dies geschieht ebenfalls für jeden Wert des Arrays „rtime“.

Während der Entwicklung dieses Algorithmus, können Probleme bezüglich der Rechnung von Lauf- bzw. Wartezeiten, aufgrund logischer Denkfehler, auftreten. Um diese Fehler zu lokalisieren wurde ein Programm implementiert, welches die Eingabe des Benutzers, sowie die Zwischenergebnisse der verschiedenen Rechnungen ausgeben kann. Anhand dieser Ausgabe kann der Benutzer nachprüfen, wo der Fehler unterlaufen ist, bzw. welche Rechenschritte im Hintergrund ablaufen. Damit der eigens entworfene Debugger (Z. 119-129) jedoch nur Zwecks der Entwicklung zur Aktion gebracht werden soll, ist dieser mit einem IF-Statement verknüpft. So muss also die Bedingung zutreffen, sodass der Debugger ausgeführt werden kann. Dieser findet mehrfache Verwendung im vorgelegten Bash-Skript.

Zu guter Letzt erfolgt die Ausgabe der Ergebnisse. Um eine tabellarische Form der Ausgabe zu ermöglichen, musste eine weitere Anweisung dafür implementiert werden. Durchsetzbar ist dies, durch die Anwendung des PRINTF-Statements, siehe **Abbildung 4**, welches für eine formatierte Ausgabe sorgt. Dabei sind verschiedene Parameter, für die Ausgabe von Integer und Strings verantwortlich.

```
133 # Tabellarische Ausgabe der Auswertung, "printf" sorgt für eine formatierte Ausgabe
134 echo -e "\n\nNun folgt eine Tabelarische Auswertung der verabeiteten Prozesse, die zuvor eingegeben wurde
135 sleep 0.4
136 printf "\n\n%-7s\t%-17s\t%-17s\t%-19s\t%-14s\n" "Prozess" "CPU-Laufzeit (ms)" "Ankunftszeit (ms)" "Gesamt
137
138 for index in ${!rtime[@]}
139 do
140                 printf "%7d\t%17d\t%17d\t%19d\t%14d\n" "${index}" "${rtime[${index}]}" "${atime[${index}]}
141 done
```

Abbildung 4: Formatierte Ausgabe des Ergebnisses durch PRINTF-Statement

Nach Fertigstellung des FIFO-Scheduling-Verfahrens, besteht die Aufgabe nun darin, das Round-Robin Verfahren auf dessen Basis zu implementieren. Bevor es daran geht den Code zu schreiben, muss man sich Gedanken um die Funktion des Algorithmus machen. Dort gibt es verschiedene Fälle zu berücksichtigen. Zu einem, wenn die CPU-Laufzeit des Prozesses größer als das Quantum ist, zum anderen der Fall, wenn die CPU-Laufzeit kleiner als das Quantum, jedoch größer als 0 ist, damit der Prozess terminiert.

Der Code für die Eingabe der Prozessanzahl, die CPU-Laufzeiten der Prozesse und dem Debugger sind äquivalent zum FIFO-Verfahren. Zunächst gilt es, das Zeitquantum für das Round-Robin-Verfahren festzulegen. Die Größe des Zeitquantums wird durch den Benutzer festgelegt (Z. 43). Nun kommt es zur Berechnung der Lauf- und Wartezeiten der einzelnen Prozesse, unter Berücksichtigung des Zeitquantums. Die zuvor definierten Elemente des Arrays „rtime“ wurden an das neue Array „openRUNS“ (Z. 51-59) übergeben. Jeder Index des Arrays „openRUNS“ wird durch eine FOR-Schleife durchlaufen (Z. 67). Im ersten Durchlauf wird der Wert an der Indexstelle 0 minus das Quantum gerechnet, welcher dann als neuer Wert an der Indexstelle 0 des Arrays „openRUNS“ zurückgegeben wird (Z. 71).
Im Anschluss folgt die Implementierung eines IF-Statements (Z. 74), welches prüft, ob der Wert an der Indexstelle 0 größer gleich dem Quantum ist. Ist dies der Fall, wird das Quantum2 um das Quantum erhöht.
Wenn dies nicht der Fall ist, wird der ELSE-IF-Fall (Z. 86) abgefragt. Die Bedingung ist erfüllt, sobald die CPU-Laufzeit kleiner als das Quantum und größer als 0 ist, damit der Prozess terminiert. In diesem Fall wird das Quantum2 um das Quantum erhöht und minus den Wert von „openRUNS“ an der Indexstelle 0 gerechnet. Wenn dieser Fall eingetreten ist dann wird im nächsten Durchlauf die letzte ELSE-IF-Bedingung (Z. 99) durchlaufen. Wenn der Prozess im letzten Schritt in der CPU abgeschlossen wird, dann wird Quantum2 wieder um das Quantum erhöht und minus den Wert von „openRUNS“ an der Indexstelle 0 gerechnet.
Im Array „neededRUNS“ (Z. 104) erfolgt die Speicherung der Anzahl an Durchläufe des Round-Robin-Verfahrens. Im Array „neededTIME“ (Z.106) werden die Laufzeiten der Prozesse gespeichert und in der Variablen „neededtimeCOMPLETE“ (Z. 108) werden die Laufzeiten addiert. Dies ist relevant für die spätere Berechnung der durchschnittlichen Laufzeit. Dann werden die Wartezeiten der Prozesse berechnet (Z. 110). Im nächsten Schritt werden in der Variablen „waitingtimeCOMPLETE“ (Z. 112) die Wartezeiten der Prozesse aufaddiert. Im Array „runcompleted“ (Z. 113) wird an der Indexstelle 0 der Wert „yes“ gespeichert. Nach jedem erfolgreichen Schleifendurchlauf wird die Variable „runCOUNT“ um eins erhöht. Nur einer der erläuterten Bedingung kann während dem Durchlauf einer Schleife erfüllt sein. Die WHILE-Schleife läuft solange, bis die Anzahl der Elemente im Array „runcompleted“ äquivalent zu der Anzahl der Elemente im Array „rtime“ ist.

Für eine tabellarische Form der Ausgabe, findet hier, wie auch schon im FIFO-Verfahren, das PRINTF-Statement seine Anwendung.

```
290 # Berechnung der durchschnittlichen Laufzeit aller Prozesse
291 averageRUNTIME=$(( ${neededtimeCOMPLETE} / ${#rtime[@]} ))
292 # Berechnung der durchschnittlichen Wartezeit aller Prozesse
293 averageWAITINGTIME=$(( ${waitingtimeCOMPLETE} / ${#rtime[@]} ))
294 # Ausgabe der durchschnittlichen Laufzeit aller Prozesse
295 sleep 0.4
296 echo -e "\n\nDurchschnittliche Laufzeit der ${#rtime[@]} Prozesse: ${averageRUNTIME} ms\n\n"
297 # Ausgabe der durchschnittlichen Wartezeit aller Prozesse
298 sleep 0.4
299 echo -e "\nDurchschnittliche Wartezeit der ${#rtime[@]} Prozesse: ${averageWAITINGTIME} ms\n\n"
```

Abbildung 5: Ausgabe und Berechnung der durchschnittlichen Lauf- & Wartezeiten

Zur Berechnung der durchschnittlichen Lauf- und Wartezeiten werden die zuvor aufeinander addierten Lauf- und Wartezeiten durch die Gesamtanzahl der Elemente im Array „rtime“ dividiert. Selbes geschieht beim FIFO-Verfahren, zur Berechnung der durchschnittlichen Lauf- und Wartezeiten.

4 Ergebnisdokumentation der Anwendung

Das Programm kann über die Kommandozeile auf jedem Linux-System aufgerufen werden: `./scheduling_simulator.bash`

Wie in **Abbildung 6** zu sehen ist, kann der Nutzer hier auswählen welches Verfahren er simulieren möchte, dazu muss er die (1) für das FIFO-Verfahren oder die (2) für das Round-Robin-Verfahren über die Tastatur eingeben.

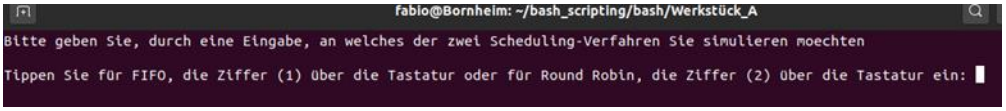

Abbildung 6: Auswahl des Verfahrens

4.1 Ergebnisse des FIFO-Scheduling-Verfahrens

Wie in **Abbildung 7** dargestellt, gibt der Benutzer an wie viele Prozesse in der CPU abgearbeitet werden sollen, Daraufhin muss der Benutzer für jeden Prozess eine Laufzeit definieren, dasselbe macht dieser auch für die Ankunftszeiten der Prozesse. Nach der Eingabe der Lauf- und Ankunftszeiten folgt eine tabellarische Ausgabe der berechneten Gesamtlauf- und Wartezeiten. Als letztes werden die durchschnittlichen Lauf- und Wartezeiten ausgegeben.

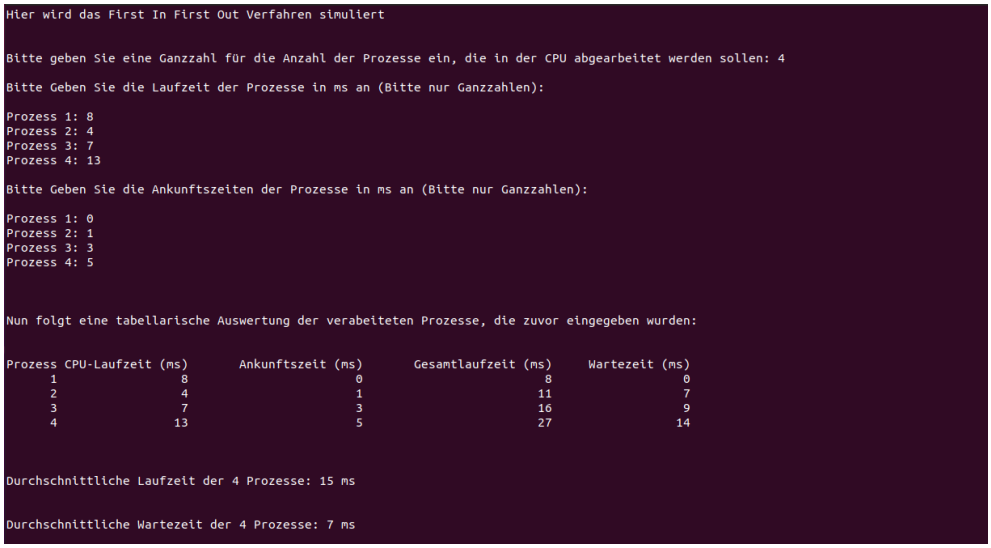

Abbildung 7: Ausgabe der FIFO-Simulation

4.2 Ergebnisse des Round-Robin-Verfahrens

Wie in **Abbildung 8** dargestellt, gibt der Benutzer an wie viele Prozesse in der CPU abgearbeitet werden sollen, Daraufhin muss der Benutzer für jeden Prozess eine Laufzeit und er muss ein Zeit-Quantum definieren. Nach der Eingabe der Laufzeit und des Zeitquantums folgt eine tabellarische Ausgabe der berechneten Gesamtlauf- und Wartezeiten. Als letztes werden die durchschnittlichen Lauf- und Wartezeiten ausgegeben.

```
Hier wird das Round Robin Verfahren simuliert


Bitte geben Sie eine Ganzzahl für die Anzahl der Prozesse ein, die in der CPU abgearbeitet werden sollen: 4

Bitte Geben Sie die Laufzeit der Prozesse in ms an (Bitte nur Ganzzahlen):

Prozess 1: 8
Prozess 2: 4
Prozess 3: 7
Prozess 4: 13

Bitte geben Sie ein Zeitquantum in ms an (Bitte nur Ganzzahlen): 1



Nun folgt eine tabellarische Auswertung der verabeiteten Prozesse, die zuvor eingegeben wurden:


Prozess CPU-Laufzeit (ms)       Quantum (ms)    Benoetigte Zyklen       Gesamtlaufzeit (ms)     Wartezeit (ms)
      1                 8                 1                 8                        26                18
      2                 4                 1                 4                        14                10
      3                 7                 1                 7                        24                17
      4                13                 1                13                        32                19


Durchschnittliche Laufzeit der 4 Prozesse: 24 ms



Durchschnittliche Wartezeit der 4 Prozesse: 16 ms
```

Abbildung 8: Ausgabe der Round-Robin-Simulation

5 Fazit

Wie bereits in der Einleitung erwähnt ist die Verwendung von Scheduling-Verfahren von elementarer Bedeutung bei der Anwendungsentwicklung. Welche Scheduling-Verfahren zur Verfügung stehen hängt von dem ausgewählten Betriebssystem und der verwendeten kernel-Version ab. Gerade bei den unterschiedlichen LINUX-Derivaten gibt es eine Vielzahl von Scheduling-Ansätzen, die einer ständigen Weiterentwicklung unterworfen sind. Hier ergibt sich ein interessantes Themengebiet, das insbesondere im Rahmen der Anwendungsentwicklung auf containerisierten Umgebungen und in der Software-Virtualisierung eine Rolle spielt.

Beim Programmieren muss sich der Entwickler im Vorfeld Gedanken machen, wie und mit welchem Scheduling-Verfahren er seine Programmieraufgabe effektiv gelöst werden kann. Für die Nutzung der Vorteile von komplexen Scheduling-Verfahren ist die Verwendung von Programmiersprachen wie Java, Python oder C (mit all seinen Varianten) eine wichtige Voraussetzung.

6 Literaturverzeichnis

Bücher & Skripte:

- Lehrmaterial zur Linux-Systemadministration und Shell-Programmierung, Christian Baun, Vorlesungsunterlagen
- UNIX-Shells, Helmut Herold, Addison Wesly, ISBN 3-89319-381-2
- Betriebssysteme kompakt, Christian Baun, Springer Vieweg (2020). 2.Auflage, ISBN: 978-3-662-61410-5

Online Literatur*:

- Shell-Programmierung, Jürgen Wolf, Rheinwerk-Verlag, ISBN 3-89842-683-1

*Alle Links wurden am 09.07.2020 überprüft.

7 Abbildungsverzeichnis