Visualisierung von Scheduling Algorithmen

Patrick Frech, Leonard Hußke, Anton Rösler

Frankfurt University of Applied Sciences
Nibelungenplatz 1
60318 Frankfurt am Main

Zusammenfassung

Die folgende Dokumentation gibt einen Überblick über Design- und Analyseentscheidungen zur Entwicklung einer Prozess-Scheduler Simulation. Das Programm wurde als Shellskript verwirklicht und simuliert die Erstellung eines Ablaufplans für Prozesse in einem Ein-Prozessor-System. Ziel ist die Visualiserung der Scheduling-Algorithmen First Come First Served (FCFS), Highest Response Ratio Next (HRRN) und Round Robin in einem Gantt-Diagramm.

Ein Prozess-Scheduler regelt die zeitliche Ausführung von Prozessen in einem Betriebssystem. Grundsätzlich unterscheidet man zwischen präemptiven und nichtpräemptiven Algorithmen. Welcher Algorithmus jedoch genutzt wird ist von den Anforderungen an den Scheduler abhängig. Um die Algorithmen vergleichen zu können muss die Performanz betrachtet werden. Diese lässt sich an der Wartezeit und Umlaufzeit von Prozessen messen.[1] Da die durchschnittliche Warte- und Umlaufzeit jedoch keine genauen Einblicke zulassen, werden die Algorithmen in einem Gantt-Diagramm visualisiert.

1 Ergebnis

Diese Dokumentation ist Teil der Portfolio-Prüfung, die im Rahmen des 2. Semesters Engineering Business Information Systems (*Wirtschaftsinformatik*) an der Frankfurt University of Applied Sciences, im Modul "*Betriebssysteme und Rechnernetze*"geleistet werden muss.

Die Implementierung aller geforderten Funktionen wurde umgesetzt und teilweise erweitert. Das Programm ist modular aufgebaut, welches das Warten bzw. Optimieren einzelner Komponenten erleichtert. Es wurde eine Menüführung implementiert, um unterschiedliche Simulationen der Algorithmen durchzuführen, ohne dabei das Programm neu starten zu müssen. Eine Beispielsimulation mit vorgegebenen Werten soll die Funktion verdeutlichen und gleichzeitig als Erklärung fungieren. Manuelle Auswahl (*Anzahl Prozesse, Ankunftszeit und Laufzeit*) ist gewährleistet und so für verschiedene Simulationen nutzbar. Eine Hilfsfunktion für alle Optionen mit möglichen Parametern wurde zusätzlich integriert.

[1] Im Laufe der Dokumentation werden auch die englischen Bezeichnungen *Waiting Time* und *Turnaround Time* verwendet, um einen näheren Bezug zum englischen Quelltext herzustellen.

Das Programm läuft fehlerfrei, ist plattformunabhängig und die Erweiterung des Programmes ist problemlos möglich.

2 Struktur

Ein Programm würde ohne den Einsatz besonderer Strukturierung unübersichtlich oder unwartbar werden. Um diese Problematik zu minimieren wird in folgendem Abschnitt der Grundaufbau des Programmes erläutert.

2.1 Aufbau

Der modulare Aufbau ermöglicht eine gute Wartbarkeit und macht den Code einfacher Zugänglich. Dazu ist der Code in unterschiedliche funktionelle Bereiche in der Verzeichnisstruktur eingeteilt.

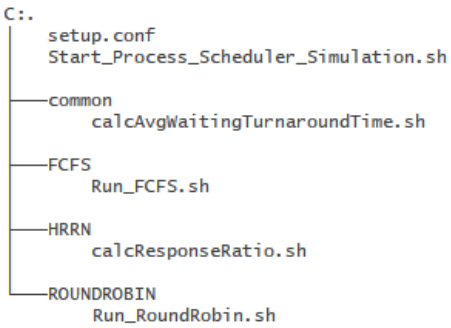

Abbildung 1. Ausschnitt aus der Verzeichnisstruktur

Das Startskript und die Konfigurationsdatei befinden sich im Basisverzeichnis des Programmordners. Die Skripte für FCFS, HRRN und RoundRobin sind in den gleichnamigen Ordnern zu finden. Hierbei ist das auszuführende Skript durch die Schreibweise *Run_JeweiligerAlgorithmus.sh* gekennzeichnet. Um Unterfunktionen deutlich abzugrenzen sind diese in der ‚camelCase'-Notation gespeichert. Ein Beispiel hierfür ist das Skript *calcResponseRatio.sh*. Weitere Funktionen die entweder mehrfach oder vom Startskript aufgerufen werden, sind in dem Ordner *common* abgelegt.

2.2 Menüführung

Da das Programm mehrere Funktionen realisiert, war eine Menüführung zwingend erforderlich. Diese sollte schlicht und logisch aufgebaut sein. Nach Start des Programms wird der Nutzer gefragt, ob dieser Beispiel-Prozesse nutzen oder

eigene Prozesse erstellen möchte. Darauf folgt ein Überblick über die verwendeten Prozesse und der Nutzer kann entscheiden, welches Scheduling-Verfahren angewandt werden soll. Dem Nutzer ist es nach jeder Simulation möglich einen anderen Algorithmus zu wählen, oder das Programm zu beenden. Zusätzlich kann vom Nutzer in diesem Menü eine Anpassung der Prozesse vorgenommen werden. Einen genaueren Überblick verschafft das Aktivitätendiagramm in Abbildung 2.(2, 135)

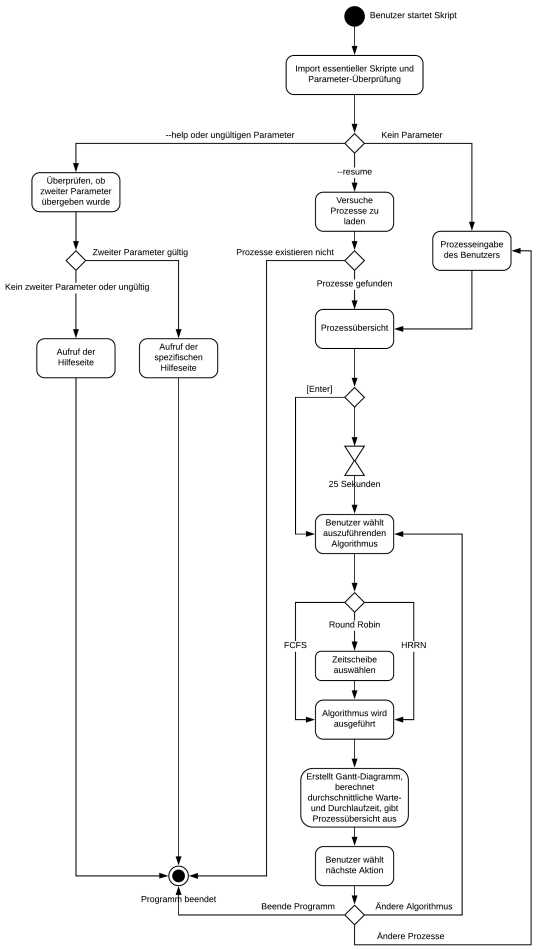

Abbildung 2. Aktivitätendiagramm des Programms

2.3 Hilfe Funktion

Das Programm wurde mit einer Hilfsfunktion ausgestattet, welche mit dem Kommandozeilenparameter `--help` aufgerufen wird. Nachfolgend die Ausgabe von `Start_Process_Scheduler.sh --help`:

```
Usage:
Start_Process_Scheduler_Simulation.sh [opt:param1]
[opt:param2]

param1 is optional and can be one of the described
commands. param2 is only used with param1=help and
has to be a valid string.

--resume          - Starts the program with the
                    processes from the last session.
--help            - Prints out this help message.
--help [param2]   - Starts the help with the given
                    param2 for specific algorithm.

Valid param2's are:

--FCFS
--HRRN
--RoundRobin
--resume
--chart
```

Anhand dieser Hilfe ist es dem Nutzer möglich, alle Funktionen des Programms zu nutzen und sich ggf. zusätzliche Informationen zu beschaffen. Nachfolgend die Ausgabe mit den beiden Parametern `--help --FCFS`:

```
Start_Process_Scheduler_Simulation.sh --help --FCFS
#########################################################
#               Process Scheduling Simulator            #
#########################################################

First Come First Served (FCFS)

FCFS is a non-preemptive scheduling algorithm. FCFS
simply queues processes in the order that they arrive
in the ready queue. The process that comes first will
be executed first and the next process only starts
after the previous process was finished.
```

Diese Information dient rein zur Beschreibung des Algorithmus und ist nicht als Funktionsbeschreibung gedacht. Das Programm setzt Vorkenntnisse im Bereich *Scheduling* voraus. Weitere Informationen zu den Algorithmen und deren detaillierter Ablauf kann unter anderem Wikipedia entnommen werden. Ferner liefert der Suchbegriff FCFS, HRRN oder RoundRobin über eine der gängigen Suchmaschinen zusätzliche umfangreiche Lektüre.

2.4 Wiederaufnahme Funktion

Die zuletzt eingegebenen Prozesse werden in der lokalen Datei *.processes.log* persistent gespeichert. Beim Aufrufen des Programmes mit dem Kommandozeilenparameter `--resume` lädt das Programm die gespeicherten Prozesse und über-springt die Prozesseingabe. So wird gewährleistet, dass der Benutzer bei Bedarf die Prozesse aus der vorherigen Sitzung nutzen kann, ohne diese neu eingeben zu müssen.

3 Implementierung

Im Folgenden wird die Implementierung angesprochen. Wenn hier die Rede davon ist, dass ein Prozess *wartet* oder *ausgeführt* wird, dann ist damit nicht gemeint dass dies tatsächlich mit einem Prozess passiert. Viel mehr ist damit gemeint, dass dies an jener Stelle simuliert wird. [2]

Da es sich nur um Scheduling-Verfahren ohne die Berücksichtigung von Prioritäten handelt, wurde bewusst auf das Implementieren eines Leerlaufprozesses verzichtet. Es ist jedoch sehr wohl möglich, dass zu einer Zeit kein bereit stehender Prozess existiert. In einer solchen Situation wird in der Simulation kein Prozess ausgeführt, d.h. im Gantt-Diagramm ist zu diesem Zeitpunkt dann eine Lücke zu sehen.[3] Das Diagramm darf gerne so interpretiert werden, dass zu einem solchen Zeitpunkt der Leerlaufprozess aktiv ist.

3.1 Grundstruktur zur Speicherung von Prozessinformationen

Damit der Benutzer einen beliebige Menge von Prozessen in der Simulation erstellen kann, werden die Parameter der Prozesse in dynamischen Arrays gespeichert. Über einen Prozess gibt es folgende Informationen:

- Prozessname
- Laufzeit
- Ankunftszeit

[2] Programmintern wird ein Prozess mit Arrays dargestellt und hat nichts mit richtigen Systemprozessen zu tun.

[3] Siehe die ersten drei Zeiteinheiten in Abbildung 4 auf Seite 9.

Für jeden der genannten Punkte wird ein eigenes Array angelegt. Dabei gehören immer die Daten mit dem selben Index im jeweiligen Array zueinander.

Tabelle 1. Vom Benutzer gewählte Parameter oder Beispiel-Prozesse

Prozessname	**Laufzeit**	**Ankunftszeit**
Pa	11	3
Pb	7	13

Folgendermaßen werden die Informationen im Programm gespeichert:

process_names = [Pa, Pb]
burst_time = [11, 7]
arrival_time = [3, 13]

Im weiteren Verlauf wird ein Array mit den Prozess IDs automatisch erstellt. Dieses enthält lediglich die Indexe. Bei *n*-Prozessen, enthält dieses Array die Zahlen von *0* bis *n-1*. Es dient dazu durch die o.g. Arrays einfacher zu iterieren und jeden Prozess eindeutig identifizieren zu können.

Es gibt kein Array in dem die Zustände der Prozesse direkt gespeichert werden. Vielmehr werden die Zustände wenn nötig berechnet, so gilt ein Prozess als bereit, wenn *Prozess Ankunftszeit - Vergangene Zeit* kleiner gleich null ist. Ein Prozess gilt dann als beendet, wenn seine *(Rest-)Laufzeit* gleich null ist.

3.2 Scheduling Simulation

Die vom Benutzer angegebenen Prozesse werden in der Simulation mit dem vom Benutzer gewählten Scheduling-Verfahren in die richtige Reihenfolge gebracht. Der jeweilige Algorithmus befüllt das Array *process_flow* in welchem an jedem Index *n* die ID des Prozesses, welcher zur Zeiteinheit *n* ausgeführt wurde, steht. Soll zu einer Zeiteinheit *n* kein Prozess ausgeführt werden wird am Index *n* des *process_flow*-Arrays eine *-1* gespeichert.

[-1 -1 -1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1]

process_flow-Array für die genannten Prozesse aus Tabelle 1 mit dem FCFS-Algorithmus geordnet. Dabei besitzt Prozess *Pa* die ID 0 und *Pb* die ID 1.

Bevor die Simulation startet, werden von den beiden Arrays *burst_time* und *arrival_time* die Kopien *bt* und *at* erstellt. Dies dient dazu, die Arrays im Laufe der Simulation verändern und daraufhin die Prozesse erneut benutzen zu können. Im *bt*-Array sind während der Simulation die noch Rest-Laufzeiten der Prozesse gespeichert, welche während den Schleifendurchläufen verringert werden. Auch die Ankunftszeiten in *at* werden nach dem ein Prozess beendet ist, verändert.

Jeder Algorithmus durchläuft eine *while*-Schleife. Die Abbruchbedingung der Schleife ist dann gegeben, wenn jeder Prozess keine verbleibende Laufzeit mehr hat:

$$\sum_{x \in bt} x = 0$$

Für jeden Algorithmus wird eine Zählervariable *clock=0* initialisiert, um die Iterationen der *while*-Schleife zu verfolgen. Jeder Erhöhung der Zählervariable ist gleichzusetzen mit dem Vergehen einer Zeiteinheit (Time Unit).

3.2.1 First Come First Served

Innerhalb der *while*-Schleife wird die Funktion *findSmallestValue* aufgerufen, welche den Index des Prozesses mit der kleinsten Ankunftszeit zurück gibt und diesen in der Variable *id* speichert. Darauf folgt eine *if*-Kontrollstruktur, die prüft, ob die Ankunftszeit des Prozesses kleiner oder gleich der aktuellen Zeit ist. Wenn dies nicht der Fall ist, dann wird dem Array *process_flow* an der Stelle *clock* eine -1 hinzugefügt. Die -1 gibt an, dass zum Zeitpunkt *clock* kein Prozess ausgeführt worden ist. Die Zählervariable *clock* erhöht ihren Wert um 1 und die *while*-Schleife beginnt die nächste Iteration.
Wenn jedoch die Ankunftszeit kleiner gleich der aktuellen Zeit ist, dann iteriert eine *for*-Schleife durch die Laufzeit (*Burst Time*). Die aufgerufene Funktion *getAllWaitingJobs* iteriert durch alle Prozesse und speichert in dem Array *isWaiting* eine 1, wenn der Prozess darauf wartet ausgeführt zu werden oder eine 0 wenn das nicht der Fall ist. Dann folgt eine weitere *for*-Schleife, die durch das Array *isWaiting* iteriert. Wenn der jeweilige Prozess nicht der Variablen *id* entspricht, dann wird die Wartezeit für den Prozess um den Wert (0 oder 1) aus *isWaiting* erhöht. Dieser Vorgang wird wiederholt, bis bei jedem Prozess die Wartezeit erhöht worden ist. Außerhalb der *for*-Schleife wird dem Array *process_flow* an der Stelle *clock* der Wert der Variable *id* hinzugefügt. Das bedeutet, dass zum Zeitpunkt *clock* der Prozess *id* ausgeführt wurde. Nachfolgend wird die Zählervariable *clock* um 1 erhöht. Nach beenden der *for*-Schleife wird die Laufzeit des durchgeführten Prozesses auf 0 gesetzt. Die Ankunftszeit wird auf einen maximalen Wert erhöht und die Umlaufzeit für den Prozess wird berechnet.
Dieser Vorgang wird so oft wiederholt, bis die Laufzeiten aller Prozesse auf 0 gesetzt sind. Das bedeutet, dass jeder Prozess ‚ausgeführt' wurde.

3.2.2 Highest Response Ratio Next

Zu Beginn jeder Iteration der *while*-Schleife wird eine Funktion aufgerufen in der die Response Ratio für alle bereitstehenden Prozesse berechnet wird. Die Funktion gibt dann die ID des Prozesses mit der höchsten Response Ratio zurück. Die weitere Verarbeitung erfolgt gemäß First Come First Served.

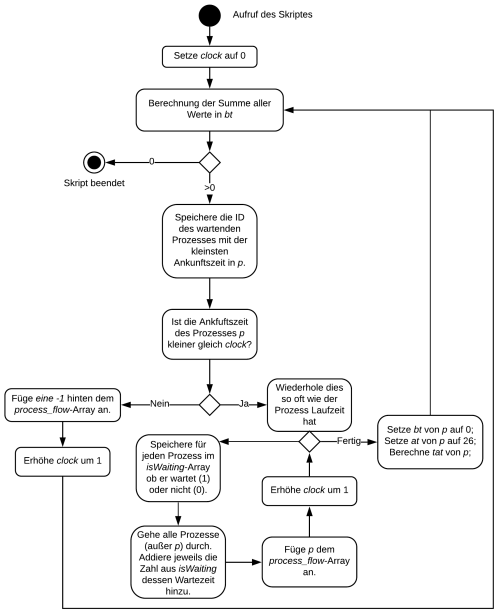

Abbildung 3. Aktivitätendiagramm von First Come First Served

3.2.3 Round Robin

Der Benutzer hat bereits nach der Auswahl des Algorithmus die Länge der Zeitscheibe festgelegt. Die Implementierung von Scheduling nach Zeitscheiben gestaltet sich zunächst sehr einfach. Jedoch entstehen Schwierigkeiten bei der richtigen Einordnung des laufenden Prozesses nach einer Zeitscheibe, wenn währenddessen weitere Prozesse eintreffen. Die Lösung ist eine Funktion *makeOrder*. Die Funktion setzt alle Prozesse, die zum Zeitpunkt des Aufrufes bereit sind und sich nicht bereits in der Warteschleife befinden, in die Warteschleife. Die Funktion erwartet beim Aufruf einen ganzzahligen Wert als Übergabeparameter. Wenn der übergebene Parameter einer ID eines bereiten Prozesses ist, wird dieser bei der Einordnung nicht berücksichtigt.

Zu Beginn wird die Funktion *makeOrder* aufgerufen, um alle Prozesse mit einer Ankuftszeit von Null in die Warteschleife zu setzen. Dann startet die Haupt-*while*-Schleife. In dieser wird der erste Prozess aus der Warteschleife geholt und gleichzeitig dort gelöscht. Der Prozess wird dann über die Länge der Zeitscheibe (oder die Laufzeit des Prozesses, sollte diese kürzer sein) ausgeführt. Bei jedem Schritt dieser Ausführung *(Zeiteinheit += 1)* wird die Funktion *makeOrder* mit der Prozess ID des gerade laufenden Prozesses aufgerufen. So wird sichergestellt, dass wenn mehrere Prozesse während des Durchlaufens einer Zeitscheibe eintreffen, diese richtig in der Warteschleife eingeordnet werden. Es werden zusätzlich

bei jedem Schritt die wartenden Prozesse berechnet und diesen im Array für die Wartezeiten die Wartezeit um *1* erhöht. Wenn nach der Zeitscheibe die Restlaufzeit noch größer als Null sein sollte, wird die Prozess ID wieder zur Warteschleife hinzugefügt. Am Ende der Iteration wird noch die Laufzeit für den Prozess berechnet, diese wird überschrieben, wenn der Prozess erneut eine Zeitscheibe bekommt.

3.3 Visualisierung

Die Visualisierung erfolgt als Gantt-Diagramm. Zu Beginn wird für jede Prozess ID ein leerer String in einem Array (*output_data*) erstellt. An den Anfang jedes Strings wird der Name des Prozesses mit der jeweiligen ID geschrieben (Auf die ersten 5 Zeichen beschränkt). Im Folgenden wird durch das *process_flow*-Array mit *p* iteriert. Es kann zwei Fälle geben:

1. Sollte *p* gleich *-1* sein, bekommt jeder String in *output_data* einen hellen Block angehängt.

2. Im anderen Fall, in dem *p* gleich einer Prozess ID ist, wird dem String im *output_data*-Array am Index *p* ein dunkler Block angehängt. Das bedeutet dass *p* zu diesem Zeitpunkt ausgeführt wird. Es wird dann mit *p2* durch alle Prozess IDs ungleich *p* iteriert. Es wird dann unterschieden. ob *p2* wartet oder nicht. Wenn *p2* wartet, wird ein halb dunkler Block an den String im *output_data*-Array an Index *p2* angehängt. Wenn *p2* nicht wartet, wird ein heller Block angehängt.

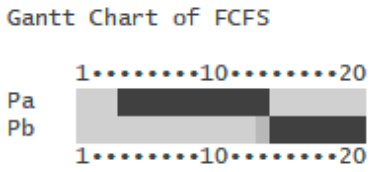

Abbildung 4. Gantt-Diagramm mit Zeitleiste

Das Diagramm wird dann mit einer dynamischen Zeitleiste ausgegeben. Zudem werden mit einem weiteren Skript die durchschnittliche Warte- sowie Laufzeit berechnet und ausgegeben. Für mehr Details wird mit einem weiteren Skript eine Tabelle erstellt welche für jeden Prozess alle gegebenen Informationen sowie die Warte- und Umlaufzeit enthält.(1)

4 Projektabschluss

Das Projekt wurde zum 24.05.2020 erfolgreich beendet. Sämtliche Funktionen wurden unter Berücksichtigung der gegebenen Testumgebung auf volle Funktionalität getestet. Ein komplett reibungsloser Ablauf auf anderen Plattformen kann aufgrund fehlender Testumgebungen nicht vollständig gewährleistet werden. Da dieses Projekt nur zu internen Zwecken (*Portfolio-Prüfung*) dient und nicht zur Veröffentlichung gedacht ist, wurde dieser Punkt bewusst nicht weiter betrachtet. Es darf allerdings davon ausgegangen werden, dass es auch auf anderen Konsolen und Betriebssystemen lauffähig ist.

5 Anhang

Nachfolgend noch eine Übersicht der verwendeten Software und der genutzten Literatur für das Projekt. Für die Shell kamen die vorinstallierte Konsole von MacOS bzw. die mitgelieferte Konsole der Github Developer Tools zum Einsatz.

5.1 Verwendete Software

Für die Umsetzung des Programms wurde auf folgende Software zurückgegriffen:

- Sublime Text 3
- Lucidchart
- Bash Shell V. 3.2.57 (*MacOS*)
- Bash Shell V. 5.0.17 (*MacOS*)
- Bash Shell V. 4.4.23 (*Windows*)
- GitHub Developer Tools (*2.24.2)*
- Xcode Developer Tools

Literaturverzeichnis

[1] Christian Baun. „5. Foliensatz Betriebssysteme und Rechnernetze„. Zugriff: 20.05.2020.

[2] Bernd Oestereich. *„Analyse und Design mit UML 2.1„, 8. aktualisierte Auflage.* Oldenbourg Wissenschaftsverlag GmbH, München, 2006.