Duale Hochschule Baden-Württemberg Mannheim

Integrationsseminar

OS Scheduling Algorithmen

Studiengang Wirtschaftsinformatik

Studienrichtung Data Science

Verfasser:	Jannik Völker, Benedikt Prisett, Eric Echtermeyer
Matrikelnummer:	–TODO–, –TODO–, 6373947
Kurs:	WWI-21-DSA
Studiengangsleiter:	Prof. Dr.-Ing. habil. Dennis Pfisterer
Dozent:	Prof. Dr. Maximilian Scherer
Bearbeitungszeitraum:	13.11.2023 – 07.02.2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

OS	Betriebssystem
CPU	Central Processing Unit
FCFS	First-Come-First-Serve
FIFO	First-In-First-Out
MLQ	Multilevel-Queue

1 Grundlagen der Visualisierung

Womöglich interessante Elemente: - Farbgebung, welche Farben sollte man wann nehmen, wie etwas darstellen - Bewegung von Objekten, wann sollte sich was wie bewegen. Eher von links nach rechts oder andersrum? - Konventionen einhalten, im Uhrzeigersinn, Titel oben links, ... - Möglichst verwandt aussehen

2 OS Scheduling Algorithmen

Das Kernstück eines jeden modernen Betriebssystems is dessen Fähigkeit eine Vielzahl von Prozessen effizient und effektiv zu verwalten. Diese Prozessverwaltung, auch bekannt als Scheduling, ist eine komplexe Aufgabe, welche darüber entscheidet, welcher Prozess zu welchem Zeitpunkt von der Central Processing Unit (CPU) verarbeitet wird. Da moderne Betriebssysteme stets eine hohe Anzahl von Hintergrundprozessen bis hin zu anspruchsvollen Anwendungen verarbeiten müssen, ist die Verwendung leistungsfähiger OS Scheduling Algorithmen essentiell. Im folgenden werden drei unterschiedliche Algorithmen des OS Scheduling vorgestellt, mit aufsteigender Komplexität. Jeder dieser Algorithmen hat eigenen Stärken und Schwächen, die ihn für bestimmte Szenarien und Anforderungen geeignet machen. Von den einfachen, aber grundlegenden Ansätzen wie First Come First Serve bis hin zu komplexeren Strategien wie Multilevel Queue Scheduling, spiegelt die Entwicklung dieser Algorithmen die Fortschritte in der Computertechnologie und unser zunehmendes Verständnis von effizientem Prozessmanagement wider.

2.1 First-Come-First-Serve

Einer der grundlegenden Scheduling Algorithmen für Betriebssysteme ist First-Come-First-Serve (FCFS), welcher auch als First-In-First-Out (FIFO) bekannt ist. FCFS verarbeitet eingehende Prozesse in der Reihenfolge ihres Eintreffens, wobei der zuerst ankommende Prozess als erstes prozessiert wird. Implementiert wird FCFS für gewöhnlich als Warteschlange, aus welcher eingehende Prozesse anschließend sequentiell verarbeitet werden können.

Der Pseudocode in Abbildung 1 implementiert den FCFS-Scheduling-Algorithmus für einen Satz von Prozessen. Dabei wird angenommen, dass jeder Prozess Eigenschaften wie Ankunftszeit (arrival) und Ausführungszeit (burst) hat. Der Algorithmus berechnet die Startzeit, Endzeit, Wartezeit und Umlaufzeit für jeden Prozess.

Der große Vorteil von FCFS liegt in dessen Einfachheit und der hieraus resultierenden leichten Implementierbarkeit. Daher wird dieser auch oft in Lehrbüchern im Kontext grundlegender Betriebssystemkonzepte diskutiert. Zudem ist FCFS transparent und einfach vor-

Algorithmus 1 FCFS Scheduling Algorithm

```
1: procedure FCFS(processes)
2:     n ← length(processes)
3:     Sort processes by arrival time
4:     for i ← 1 to n do
5:         if i = 1 then
6:             start_time[i] ← processes[i].arrival
7:         else
8:             start_time[i] ← max(processes[i].arrival, finish_time[i − 1])
9:         end if
10:        finish_time[i] ← start_time[i] + processes[i].burst
11:        waiting_time[i] ← start_time[i] − processes[i].arrival
12:        turnaround_time[i] ← finish_time[i] − processes[i].arrival
13:    end for
14:    return start_time, finish_time, waiting_time, turnaround_time
15: end procedure
```

hersehbar, da die Reihenfolge und Bearbeitungsdauer aller Prozesse lediglich von deren Ankunftszeiten abhängig ist. Ein zusätzlicher Vorteil liegt in der fairen Behandlung aller Prozesse, welche ohne Bevorzugung stattfindet, da jeder Prozess in der Reihenfolge seines Eintreffens bearbeitet wird.

Nichtsdestotrotz, weist FCFS auch signifikante Nachteile auf, weshalb in der Praxis meist von einer alleinigen Nutzung dieses Algorithmus abgesehen wird. Das wesentliche Problem ist nämlich der Convoy-Effekt, bei dem ein langer Prozess, der früh in der Warteschlange erscheint, nachfolgende, kürzere Prozesse verzögert. Diese Situation führt zu einer ineffizienten CPU-Auslastung und verlängerten Wartezeiten. Weiterhin berücksichtigt FCFS neben der Dauer auch nicht die unterschiedliche Priorität von Prozessen, was besonders nachteilig für interaktive Systeme ist, in denen schnelle Antwortzeiten von höchster Relevanz sind. Diese Mängel machen FCFS für viele moderne Anwendungen unpraktikabel. Daher wird im folgenden der OS Scheduling Algorithmus Round Robin näher betrachtet, welcher versucht eine schnellere Antwortzeit zu ermöglichen.

2.2 Round Robin

Round Robin ist ein weit verbreiteter OS Scheduling-Algorithmus in Betriebssystemen, welcher für seine Fairness und Eignung für Zeitscheiben-basiertes Multitasking bekannt ist.

Dieser Algorithmus weist jedem Prozess in der Warteschlange ein festes Zeitintervall, auch bezeichnet als Quantum, zu. Nach Ablauf des Quantums wird der aktuell laufende Prozess unterbrochen und an das Ende der Warteschlange gestellt, um dem nächsten Prozess in der Warteschlange CPU-Zeit zuweisen zu können. Diese Methode gewährleistet, dass alle Prozesse regelmäßige CPU-Zeit erhalten und kein Prozess andere blockiert, wie es bei FCFS mit dem Convoy-Effekt der Fall ist. Durch diese Rotation wird eine gleichmäßigere Verteilung der CPU-Zeit über alle Prozesse erreicht.

Algorithmus 2 Round Robin Scheduling Algorithmus

```
1: procedure RoundRobin(processes, quantum)
2:     n ← length(processes)
3:     time ← 0
4:     Initialize remaining_burst[n] with burst times of processes
5:     while any remaining_burst > 0 do
6:         for i ← 1 to n do
7:             if remaining_burst[i] > 0 then
8:                 start_time[i] ← time
9:                 Execute process i for min(remaining_burst[i], quantum) time
10:                time ← time + min(remaining_burst[i], quantum)
11:                remaining_burst[i] ← remaining_burst[i] − min(remaining_burst[i], quantum)
12:                if remaining_burst[i] = 0 then
13:                    finish_time[i] ← time
14:                    waiting_time[i] ← finish_time[i] − processes[i].burst − processes[i].arrival
15:                    turnaround_time[i] ← finish_time[i] − processes[i].arrival
16:                end if
17:            end if
18:        end for
19:    end while
20:    return start_time, finish_time, waiting_time, turnaround_time
21: end procedure
```

Abbildung **??** zeigt Pseudocode für die Implementation des Round Robin Scheduling Algorithmus für einen Satz von Prozessen. Jeder Prozess wird für eine Zeitdauer bis zum definierten Quantum ausgeführt. Der Algorithmus berechnet Start- und Endzeiten, Wartezeiten und Umlaufzeiten für jeden Prozess.

Ein wesentlicher Vorteil des Round Robin-Algorithmus ist dessen Fähigkeit, eine niedrige Antwortzeit für alle Prozesse zu gewährleisten, weshalb dieser besonders für interaktive Systeme von hoher Eignung ist. Da jeder Prozess innerhalb eines bestimmten Zeitrahmens

bedient wird, kommt es nicht zu langen Wartezeiten bis die Prozesse CPU-Zeit zugeteilt bekommen. Diese Eigenschaft wird von Silberschatz, Galvin und Gagne (2018) als entscheidend für Systeme mit hoher Prozessanzahl und Anforderungen an die Reaktionsfähigkeit angesehen, wie es bei modernen Betriebssystemen der Fall ist.

Allerdings weist Round Robin auch Nachteile auf. Ein wesentlicher Punkt ist hierbei die Wahl der Länge des Zeitquantums. Bei einem zu kurzen Quantum, kommt es durch den Round Robin Algorithmus zu häufigen Prozesswechseln, wodurch ein Overhead entsteht und hierdurch die CPU-Effizient verringert wird. Wenn allerdings ein zu langes Quantum gewählt wird, verlängern sich die Antwortzeiten für Prozesse und in extremen Fällen kann es zum gleichen Convoy-Effekt wie bei FCFS kommen. Daher ist die Optimierung des Quantums abhängig von den spezifischen Anwendungsumgebungen und von hoher Relevanz. Ein weiterer Nachteil, ähnlich wie bei FCFS ist die fehlende Betrachtung von unterschiedlichen Prioritäten der eingehenden Prozesse. Im folgenden wird ein weiterer OS Scheduling Algorithmus beschrieben, welcher versucht diese Herausforderung eines prioritäten-basierten Schedulings zu beheben.

Insgesamt bietet der Round Robin-Algorithmus eine ausgewogene Lösung für das Scheduling-Problem, insbesondere in Umgebungen, bei welchen Fairness und schnelle Antwortzeiten gefordert sind. Seine Einfachheit und Effizienz machen ihn zu einer beliebten Wahl in vielen Betriebssystemen.

2.3 Multilevel Queue Scheduling

Die Prozesse, welche ein modernes Betriebssystem verarbeitet kann in unterschiedliche Kategorien unterteilt werden. So gibt es beispielsweise interaktive Prozesse, bei welchen eine schnelle Antwortzeit essentiell ist, und Hintergrundprozesse, welche nicht direkt abgearbeitet werden müssen. Für gewöhnlich haben interaktive Prozesse daher eine höhere Priorität und müssen daher schneller CPU-Zeit zugeteilt bekommen. Multilevel-Queue Scheduling (MLQ) Scheduling ist ein fortgeschrittener Scheduling-Algorithmus, welcher versucht diese Prozesse mit unterschiedlichen Prozessen effizient zu verwalten. Bei MLQ wird die Prozesswarteschlange in mehrere separate Warteschlangen aufgeteilt, wobei jede Warteschlange eine eigene Prioritätsebene besitzt. Eingehende Prozesse werden nun basierend auf ihrer Priorität in die jeweilig zuständige Warteschlange eingeteilt. Jede dieser

Warteschlangen hat nun einen eigenen Scheduling-Algorithmus, um eine differenzierte Behandlung der Prozesse zu ermöglichen. Bei der Verarbeitung der Prozesse wird nun stets die Warteschlange mit der höheren Priorität zuerst abgearbeitet, bis diese vollständig entleert wurde. Anschließend fängt die Verarbeitung der nächst-geringeren Prioritätsstufe an. Sofern ein Prozess mit höherer Priorität nun eintreffen sollte, wird die Verarbeitung der Warteschlange mit geringerer Priorität pausiert.

Algorithmus 3 Multilevel Queue Scheduling Algorithmus mit FCFS and Round Robin

```
1: procedure MLQ(queues, processes, quantum)
2:     Assign each process to a queue based on its priority or category
3:     for each queue in queues do
4:         if queue is for interactive tasks then
5:             Apply Round Robin Scheduling (Refer to Round Robin Algorithm) with
   quantum quantum
6:             Execute interactive tasks in queue using RR
7:         else if queue is for background tasks then
8:             Apply First-Come, First-Served Scheduling (Refer to FCFS Algorithm)
9:             Execute background tasks in queue using FCFS
10:        end if
11:    end for
12:    return scheduling results for each process
13: end procedure
```

Der Pseudocode aus Abbildung 3 beschreibt, wie bei MLQ verschiedene Warteschlangen für interaktive und Hintergrundtasks verwendet werden, wobei für interaktive Tasks der Round Robin und für Hintergrundtasks der FCFS angewendet wird.

Der zentrale Vorteil von MLQ liegt in dessen Flexibilität und Effizienz bei der Behandlung verschiedener Prozesstypen. Beispielsweise können Systemprozesse, interaktive Prozesse und Batch-Prozesse in verschiedenen Warteschlangen mit entsprechenden Prioritäten und Scheduling-Strategien verwaltet werden. Hierdurch wird eine bessere Anpassung an die Anforderungen spezifischer Prozesstypen erreicht, was zu einer verbesserten Gesamtleistung des Systems führt.

Ein Nachteil von MLQ liegt allerdings in seiner Komplexität, sowohl in der Implementierung als auch im Management. Die korrekte Einordnung von Prozessen in Warteschlangen und die Auswahl geeigneter Scheduling-Algorithmen für jede Warteschlange erfordern sorgfältige Planung und ständige Anpassung. Eine nachteilige Auswahl und Konfiguration dieser Algorithmen kann zu einem erhöhten Overhead führen und die Systemeffizienz negativ beeinträchtigen.

Trotz dieser Herausforderungen ist MLQ ein beliebter Scheduling Algorithmus in Betriebssystemen, insbesondere dort, wo eine Vielzahl unterschiedlicher Prozesse und Anforderungen effizient verwaltet werden muss.

3 Quantitativer Vergleich

3.1 Metriken

Die richtige Auswahl dieser Algorithmen ist ein kritischer Bestandteil, um die Gesamtleistung und Reaktionsfähigkeit des Computersystems zu optimieren. Die zentrale Herausforderung hierbei ist die begrenzte Leistungsfähigkeit der Hardware, insbesondere der CPU, da diese zu einem Zeitpunkt stets nur einen Prozess ausführen kann. Dieser Engpass wird durch die OS Scheduling Algorithmen gemindert, da diese entscheiden welcher Prozess als nächstes Zugang zur CPU erhalten soll. Ein effektives OS Scheduling muss dabei eine Balance zwischen verschiedenen wichtigen Metriken finden:

- **Durchsatz (Throughput)**:

  Der Durchsatz misst die Anzahl der Prozesse, die in einer bestimmten Zeiteinheit vollständig abgearbeitet werden. Ein höherer Durchsatz bedeutet eine effizientere Verarbeitung von Prozessen durch das System. Bei vollständiger CPU-Auslastung, ist diese Metrik bei unterschiedlichen Algorithmen konstant.

  $$Durchsatz = \frac{Anzahl\ abgeschlossener\ Prozesse}{Zeiteinheit}$$

  Diese Formel berechnet den Durchsatz als Anzahl der Prozesse, die pro Zeiteinheit abgeschlossen werden.

- **Wartezeit (Waiting Time)**:

  Die Wartezeit ist die Gesamtzeit, die ein Prozess in der Warteschlange verbringt, bevor er Zugang zur CPU erhält. Niedrigere Wartezeiten sind in der Regel wünschenswert, da sie auf ein reaktionsfähigeres System hinweisen.

  $$Wartezeit = \frac{1}{n}\sum_{i=1}^{n}(Startzeit_i - Ankunftszeit_i)$$

  Hierbei ist n die Anzahl der Prozesse, und die Wartezeit wird als Durchschnitt der Zeit berechnet, die jeder Prozess vom Ankunftszeitpunkt bis zum Start der Ausführung wartet.

- **Antwortzeit (Response Time)**:

  Die Antwortzeit misst die Zeit vom Beginn eines Prozesses bis zur ersten Antwort, nicht bis zur vollständigen Ausführung. Diese Metrik ist besonders wichtig in interaktiven Systemen, wo eine schnelle Reaktion auf Benutzereingaben erforderlich ist.

  $$Antwortzeit = \frac{1}{n}\sum_{i=1}^{n}(Erste\ Antwortzeit_i - Ankunftszeit_i)$$

  Die Antwortzeit misst die Zeit vom Ankunft bis zur ersten Antwort jedes Prozesses, gemittelt über alle Prozesse.

- **Ausführungszeit (Turnaround Time)**:

  Die Ausführungszeit ist die gesamte Zeit vom Start eines Prozesses bis zu seinem Abschluss. Diese Metrik berücksichtigt sowohl die Wartezeit als auch die Ausführungszeit und gibt somit ein umfassendes Bild von der Effizienz des Scheduling-Algorithmus.

  $$Ausführungszeit = \frac{1}{n}\sum_{i=1}^{n}(Abschlusszeit_i - Ankunftszeit_i)$$

  Die Ausführungszeit wird als durchschnittliche Gesamtzeit berechnet, die ein Prozess vom Ankunftszeitpunkt bis zum Abschluss benötigt.

- **CPU-Auslastung (CPU Utilization)**:

  Diese Metrik gibt an, wie effektiv die CPU genutzt wird. Eine hohe CPU-Auslastung bedeutet, dass die CPU aktiv Prozesse bearbeitet und nicht untätig ist, was auf eine effiziente Nutzung der Ressourcen hinweist.

  $$\text{CPU} - Auslastung = \left(\frac{Gesamtzeit\ \text{CPU} - Beschäftigung}{Gesamtbeobachtungszeit}\right) \times 100\%$$

  Diese Formel gibt den Prozentsatz der Zeit an, in der die CPU aktiv Prozesse bearbeitet hat, bezogen auf die gesamte Beobachtungszeit.

- **Fairness**:

  Fairness misst, wie gleichmäßig die CPU-Zeit zwischen den verschiedenen Prozessen aufgeteilt wird. Ein fairer Scheduling-Algorithmus stellt sicher, dass kein Prozess übermäßig bevorzugt oder benachteiligt wird. Es handelt sich hierbei um eine qualitative Metrik und lässt sich nicht direkt durch eine allgemeine Formel quantifizieren.

Stattdessen wird sie oft durch die Analyse der Verteilung der CPU-Zeit und der Wartezeiten über die verschiedenen Prozesse hinweg bewertet.

3.2 Simulationsergebnisse

fehlt noch hehe

4 Anwendungsgebiete

A Quellcode

Der vollständige Quellcode ist über folgenden GitHub Link erreichbar: https://github.com/echtermeyer/OS-Scheduling-Algorithms-Comparison