Betriebssysteme 2022 – Gruppe 6

8. JANUAR

Shortest-remaining-time

Verfasst von: Karen Sardarjan
Tariq Al-Arashi

Scheduling Algorithmus – Vorwort

Gesamter Designreport als Gruppenarbeit zu benoten.

In folgender Projektaufgabe werden wir einen Scheduler für eine gegebene Testumgebung implementieren. Diese Testumgebung ist lediglich für das Ausführen unserer implementierten Lösung zuständig und für das Testen durch Prozesse die scheduled werden.

Um nun entscheiden zu können welcher Scheduler implementiert wird, haben wir uns in der Theorie mit jedem Verfahren auseinandergesetzt und haben uns am Ende für das Shortest-Remaining-Time-Next Verfahren entschieden.

Der SRTN-Algorithmus arbeitet in einer Art und Weise, bei der beim Scheduling die Verbleibende Zeit eines Prozesses als Priorität gesehen wird. Dies bedeutet, dass die Prozesse in der Priority Queue anhand der Verbleibenden Zeit sortiert werden. So würde bei 2 Prozessen A und B mit den Verbleibenden Zeiten A (100) und B (50) B priorisiert werden und demnach auch vorher ausgeführt werden. Da es sich beim SRTN-Algorithmus um einen unterbrechenden Scheduling Algorithmus handelt, kann, während ein Prozess ausgeführt wird, dieser jederzeit gestoppt werden und ein Prozess mit kürzerer Verbleibender Zeit ausgeführt werden. So wäre es möglich, dass beim Vergleich des neuen und des gerade ausgeführten Prozesses, der neue eine kürzere Verbleibende Zeit hat, der neue sofort ausgeführt und der alte gestoppt wird.

Annahmen

Als ersten Schritt haben wir uns als Ziel gesetzt, den gesamten Code in der gegebenen Testumgebung zu verstehen, um uns so ein Bild machen zu können, was genau bereits auf welche Art und Weise implementiert worden ist und auf welche Art und Weise wir die bereits implementierten Komponenten und Funktionalitäten verändern müssen, um den SRTN-Algorithmus implementieren zu können.

Da der FCFS kein multiprogramming Algorithmus ist und somit die gesamte Funktionalität nur auf einen derzeit arbeitenden Prozess ausgelegt war, der immer wieder scheduled wird, bis dieser fertig ist, war unsere erste Annahme, dass wir zunächst einmal an der Datenstruktur arbeiten müssen, bevor wir mit dem Scheduler anfangen würden. Die gegebene Readylist und die Blockedlist waren keine Liste und müssten verändert werden, um mehrere Prozesse gleichzeitig speichern zu können.
Um nun von einem primitiven Single Process Ansatz zu einem Multiprogramming System zu kommen, müssten wir annehmen, dass das gesamte Block- und Ready System zu Priority Queues verändert, werden müssen, um den SRTN-Algorithmus realitätsnah implementieren zu können.

Entwurfsentscheidungen

So haben wir uns dazu entschieden, die sowohl rechenbereiten als auch blockierten Prozesse in einer Datenstruktur die eine Linked List als Basis, als Ziel zu setzen, um eine einfache Erweiterbarkeit zu ermöglichen. Bei der Wahl der Datenstruktur für die Ready- und Blocklist haben wir uns dazu entschieden eine Priority Queue zu implementieren, da die Reihenfolge der Prozesse von großer Relevanz ist und wir durch den Head der Priority Queue, an den nächst zu schedulten oder unblockierten Prozess kommen. So ist es beispielsweise bei der Priority Queue für die rechenbereiten Prozesse so, dass Prozesse die neu gestartet werden, dynamisch auf ihre Verbleibende Zeit überprüft werden, bevor sie in die Readyqueue gesetzt werden. So erhalten wir die Möglichkeit sicherzustellen, dass Prozesse, die eine kürzere Verbleibende Zeit haben durch die Priority Queue richtig einsortiert werden und dadurch auch der kürzeste Prozess als erstes ausgewählt wird, da dieser als erster Eintrag in der Priority Queue steht.

Für die Blocklist gilt das gleiche. Der große Unterschied zur Readylist liegt daran, dass als relevante Zeit, die für die Prioritäten wichtig ist, nicht die Verbleibende Zeit ist, sondern die IOready Zeit. Die IOready Zeit ist hierbei die Zeit, wann ein Prozess nicht mehr blockiert ist und bereit ist wieder in die Ready Queue hinzugefügt zu werden. So würde jener Prozess mit kürzester IOready Zeit am Anfang der Priority Queue wiederzufinden sein. Die Priority Queues sind in unserer Implementierung das Wartesystem, da die Prozesse, die in den Priority Queues gespeichert sind, darauf warten, dass sie gescheduled werden oder unblockiert werden.

Eine Entscheidung, die wir ebenfalls treffen mussten bei der Implementierung der Priority Queues war, wie Prozesse, die die gleiche Verbleibende Zeit oder IOready Zeit haben, priorisiert werden.

in diesem Fall haben wir uns dazu entschieden, dass wenn zwei Prozesse gleichzeitig mit derselben Verbleibenden Zeit in die Ready Queue bzw. IOready Zeit in die Blocked Queue hinzugefügt werden sollen, den Prozess, der als erstes hinzugefügt wird, auch an die erste Stelle zu setzen. Dies gilt ebenfalls wenn bereits ein Prozess in der jeweiligen Priority Queue existiert und ein neuer mit derselben Verbleibenden Zeit oder IOready Zeit hinzugefügt wird. Der Grund für diese Entscheidung war der, dass wir aus Fairness, den Prozess, der in Theorie länger in der Priority Queue war und gewartet hat, auch priorisiert werden soll.

Überblick über den Funktionsumfang, Betriebsbedingungen und etwaige Limitationen

Überblick

In **Abbildung 1 Überblick über die Core** wird der gesamte Core visualisiert. Als erstes wird ausgewählt welcher Prozess ausgeführt werden soll aus der Ready Queue bzw. Priority Queue. Sollte kein Prozess in der Ready Queue sein, gehen wir in die Idle Time und suchen nach unblocked oder neu gestarteten Prozessen. Sollte ein Prozess unblocked oder gestartet werden wird dieser in die Ready Queue hinzugefügt. Wird ein Prozess ausgewählt dann wird dieser ausgeführt. Während der Ausführung dieses Prozesses wird geprüft ob zwischenzeitlich neue Prozesse gestartet worden sind oder ob Prozesse aus der Blocked Queue wieder unblockiert worden sind. Wenn das der Fall ist, dann wird dieser Prozess in die Ready Queue hinzugefügt. Nach der Ausführung beobachten wir das Scheduling Event und entscheiden anhand dessen die nächste Aktion. So muss man zwischen drei unterschiedlichen Scheduling-Ereignissen (Completed, Quantum Over, IO) unterscheiden.

- Completed steht hierbei dafür, dass ein Prozess seine Ausführung erfolgreich beendet hat und somit gelöscht werden kann.
- Beim Quantum Over wird der Prozess wieder in die Ready Queue hinzugefügt, nach dem das Quantum vollständig ausgenutzt wurde.
- IO-Aufruf steht dafür, dass ein Prozess bei der Ausführung wegen einem Input/Output Aufruf blockiert wurde und somit in die Blocked Queue hinzugefügt werden kann.

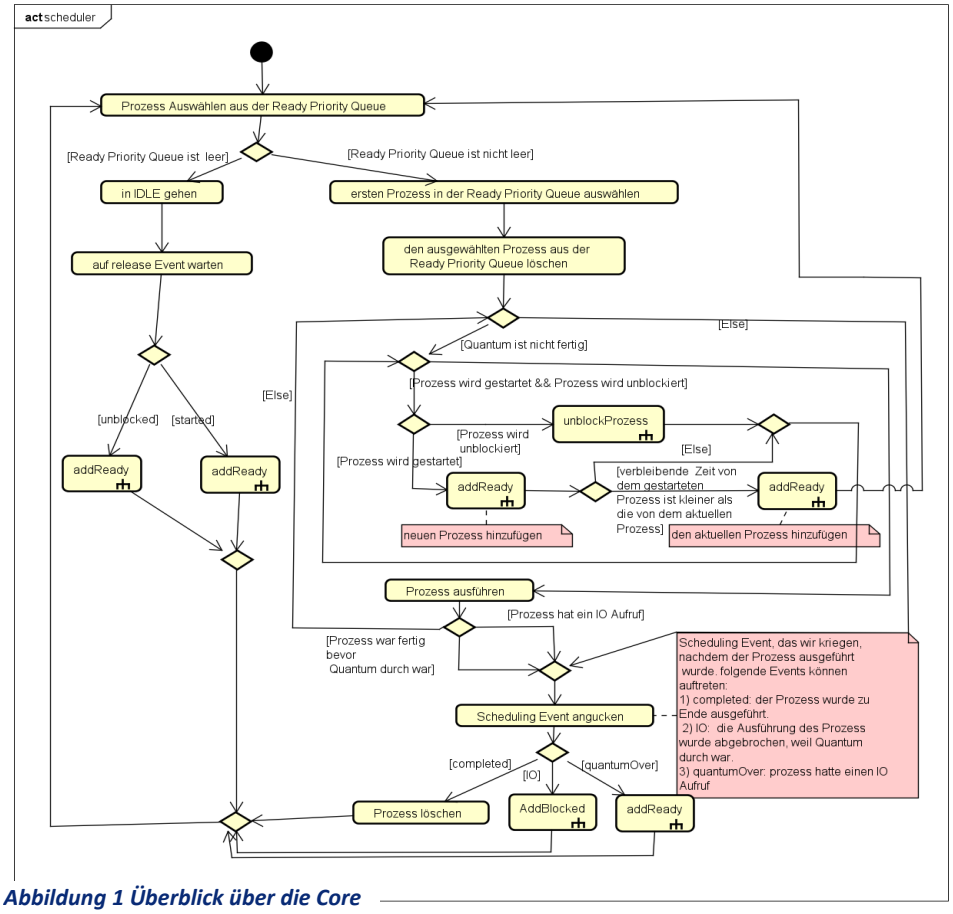

Abbildung 1 Überblick über die Core

Betriebsbedingungen und Limitationen

Limitationen

Die Limitation unserer Implementierung liegt darin, dass wenn es während des Ausführens eines Prozesses dazu kommt, dass ein Prozess gestartet wird, der eine kürzere Verbleibende Zeit als der derzeitige Prozess hat, erst bei Quantum Ende oder bei einem I/O der andere Prozess ausgewählt wird und als nächstes scheduled wird. So ist es beim Shortest Remaining Time Next Algorithmus jedoch so, dass durch das präemptive Verfahren ein Prozess nicht bis zum Ende (Quantum beispielsweise) ausgeführt werden muss sondern mittendrin gestoppt werden kann und dadurch der kürzere Prozess ausgewählt wird.
Dadurch das wir limitiert bei der Implementierung sind und weder die Simulation noch den Dispatcher manipulieren dürfen, können wir während der Ausführung eines Prozesses, diesen nicht stoppen. In dem Core wird nach der Auswahl des Prozesses der scheduled wird, die Funktion sim_runProcess ausgeführt, die sofort die Systemzeit verändert, bevor wir gestartete Prozesse beobachten können und anhand dessen Informationen Scheduling Entscheidungen treffen könnten.

Bedingungen

Die einzige Bedingung, die erfüllt sein muss, damit der Scheduler funktioniert ist die, das Prozesse existieren müssen. Diese Prozesse müssen richtig beschrieben sein mit der PID, Startzeit, Ausführungsdauer und Typ des Prozesses.

Funktionsumfang

Unser Scheduler ermöglicht das Scheduling von unendlich vielen Prozessen. So kann unser Scheduler anhand der angegebenen Daten in der Process Datei mit der Information, wann der jeweilige Prozess in Systemzeit gestartet wird und wie lange dieser ausgeführt wird, entscheiden wann dieser scheduled wird. Mittels SRT-Algorithmus wird kontinuierlich der Prozess ausgewählt und ausgeführt der die kürzeste Zeit benötigt und tut dies auch, wenn beispielsweise ein kürzerer Prozess erscheint, der derzeitige Prozess blockiert wird oder das Quantum over ist. Somit blockiert ein Prozess nie einen kürzeren.

Implementierung

Zunächst einmal haben wir die Ready Queue und Blocked Queue so verändert und neu implementiert, indem wir jeweils eine Priority Queue implementiert haben. Die Priority Queues bestehen aus Elementen, die miteinander mittels Pointer verbunden sind. So besteht ein Element aus der Ready Priority Queue aus der PID, Verbleibende Zeit und einem Pointer auf das nächste Element. Dasselbe gilt für die Block Priority Queue jedoch wird statt Verbleibende Zeit, die IOReady Zeit gespeichert.

Funktionen die vorher noch nicht existierten:

- unblockProcess (processcontrol.c)
    - Visualisiert wird in **Abbildung 2 Ablauf von unblockProcess**, der Ablauf für unblockProcess. Diese Funktion hat Zugriff auf die Blocked Priority Queue und wird genutzt, um zu überprüfen, ob das erste Element in der Priority Queue unblockiert werden kann. Sollte das der Fall sein wird das Element aus der Blocked Priority Queue gelöscht und in die Ready Priority Queue hinzugefügt

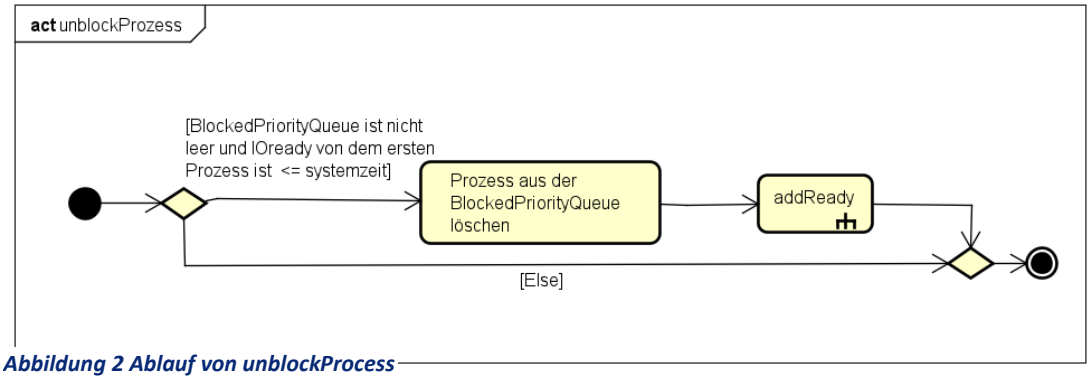

Abbildung 2 Ablauf von unblockProcess

- newReadyListElement (processcontrol.c)
    - Erzeugt neues ReadyList Element
- newBlockedListElement (processcontrol.c)
    - Erzeugt neues BlockedList Element
- logAdd (processcontrol.c)
    - Wird aufgerufen, wenn ein Prozess in die Ready Priority Queue oder in die Blocked Priority Queue hinzugefügt wird
    - Gibt Informationen über den neu gespeicherten Prozess

So haben wir folgende Funktionen geändert, die bereits existierten:

- addReady (processcontrol.c)
    - In **Abbildung 3 Ablauf von addReady** ist die funktionalität von addReady visualisiert. Wir erzeugen ein Element und Reservieren Speicher für das neue Element, setzen die PID auf die übergebene PID. Berechnen die Verbleibende Zeit mittels Duration aus der Prozesstabelle und subtrahieren davon die usedCPU. Setzen Pointer für das nächste Element erstmal auf NULL. Suchen die richtige Position für das Element, indem wir unsere Queue durchsuchen und die bestmögliche Position finden.

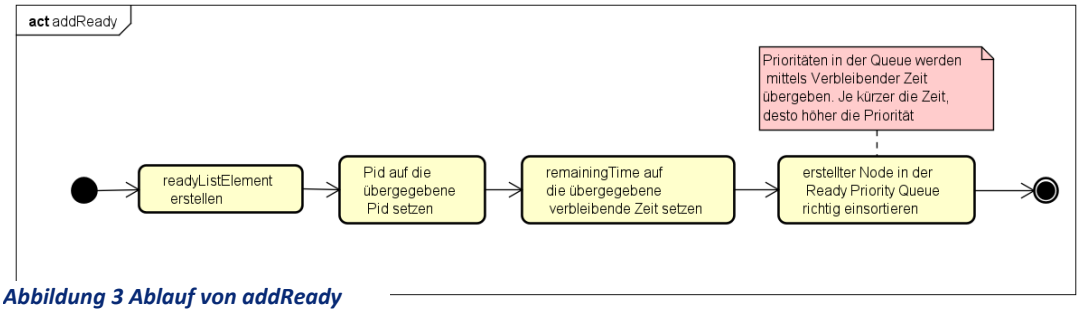

Abbildung 3 Ablauf von addReady

- removeReady (processcontrol.c)
    - Holen uns einen Pointer auf den Head der Queue und setzen diesen auf das nächste Element und setzen Speicher wieder frei
- addBlocked (processcontrol.c)
    - Visualisiert wird in **Abbildung 4 Ablauf von AddBlocked**, wie addBlocked funktioniert. Wie addReady, nur dass IOready genutzt wird und diese mittels Systemzeit + die Blockdauer ermittelt wird

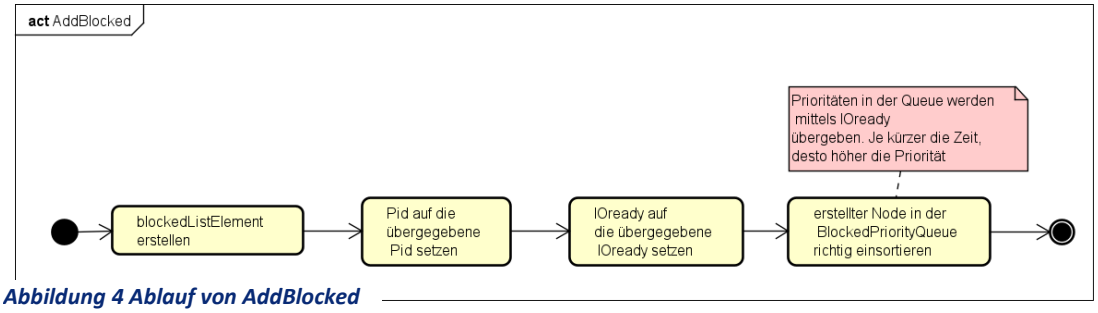

Abbildung 4 Ablauf von AddBlocked

- removeBlocked (processcontrol.c)
    - Wie removeReady, nur dass hier das blockierte Element in der Blocked Queue gelöscht wird
- Schedule (scheduler.c)
    - Element, welches ausgewählt wurde, wird sofort von der Ready Priority Queue gelöscht
- coreLoop (core.c)
    - In der Multiprogramming Loop
        - Hier werden Prozesse mittels unblockProzess unblockiert und in die Ready Priority Queue hinzugefügt.

Quellen

Priority Queue using Linked List in C (tutorialspoint.com)

Moderne Betriebssysteme – Seite 209

Vorlesungsskript Betriebssysteme - Microsoft PowerPoint - 04_Scheduling.pptx (th-luebeck.de) **– Folie 19-20**

Understanding and implementing a Linked List in C and Java - YouTube

2.10 - SRTN (Shortest Remaining Time Next) Scheduling Algorithm - YouTube