

Schedulingalgorithmen und Rechenzeitverteilung auf Betriebssystemebene

Andi Drebes

Fachbereich Informatik der Universität Hamburg

Vogt-Kölln-Straße 30

Hamburg, Germany

5drebes@informatik.uni-hamburg.de

ABSTRACT

Unter Prozess-Scheduling versteht man im Allgemeinen die Art und Weise, wie Rechenzeit in einem System an Prozesse verteilt wird. Einleitend werden die generellen Schwierigkeiten beim Design allgemein verwendbarer Schedulingalgorithmen aufgezeigt. Danach werden einige der grundlegenden, einfacheren, Schedulingalgorithmen vorgestellt und deren Vor- und Nachteile diskutiert. Der zweite Teil des Papers beschäftigt sich mit der Implementierung des $O(1)$ -Schedulers des Linux-Kernels 2.5 bzw. 2.6. Hierbei wird nicht auf Details der Implementierung eingegangen, sondern es wird die allgemeine Funktionsweise des $O(1)$ -Schedulers erläutert. I/O-Scheduling ist nicht Thema dieses Papers.

General Terms

Scheduling, Schedulingalgorithmen

Keywords

Schedulingalgorithmen, Scheduling, Rechenzeitverteilung, Round-Robin, $O(1)$ -Scheduler, Linux-Kernel, Multitasking, preemptives Multitasking, kooperatives Multitasking, Betriebssystem, First-come-first-served, shortest job first, priority scheduling, lottery scheduling, Fair-Share-Scheduling

1. EINLEITUNG

Schedulingalgorithmen sorgen in Betriebssystemen für Verteilung der Rechenzeit auf einzelne Prozesse. Auf den einfachsten Stapelverarbeitungssystemen war dies kein Problem. Der Scheduler – falls man bereits von einem solchen sprechen konnte – funktionierte auf die denkbar einfachste Weise. Nach Beendigung eines Jobs wurde lediglich der nächste ausgeführt. Das Problem des Prozess-Scheduling ergab sich aber bereits mit dem Stellen höherer Anforderungen an solche Systeme. Spätestens mit der Einführung von Time-Sharing-Systemen, auf denen die Ressource CPU-Zeit möglichst optimal auf die einzelnen Prozesse verteilt werden musste, befassten sich eine ganze Menge von Wissenschaftlern und Entwicklern mit der Entwicklung effizienter Algorithmen in diesem Bereich. Die Entwicklung immer schnellerer Rechner, die Einführung von Personalcomputern und die Entwicklung des Rechners zur multimedialen Unterhaltungs- und Arbeitsplattform verlagerte das Problem in Richtung der optimalen Verteilung der Rechenzeit im Hinblick auf möglichst geringe Reaktionszeiten interaktiver Systeme. Es werden nun ganz andere Ansprüche an Schedulingalgorithmen gestellt, da sich das Nutzungsverhalten der Benutzer solcher

Systeme verändert hat. Im Gegensatz zu früher wird ein Computer von der breiten Masse nicht mehr nur für einfache Aufgaben wie Textverarbeitung und Datenspeicherung verwendet – vielmehr werden komplexe Aufgaben wie beispielsweise das Dekodieren großer Mengen von Bild- und Videodaten darauf ausgeführt.

Dieses Paper soll einen Überblick über das Prozess-Scheduling geben. Angefangen mit der Erläuterung der einfachsten Scheduling-Strategien aus den Anfängen der Forschung, wird im Verlauf des Textes der Grad der Komplexität der einzelnen Algorithmen zunehmen. Dabei wird auf generelle Schwierigkeiten beim Design der Algorithmen hingewiesen und mögliche Lösungsansätze besprochen. Für die Implementierung eines Scheduling-Algorithmus soll exemplarisch der, in vielerlei Hinsicht interessante, $O(1)$ -Scheduler des Linux-Kernels der Reihe 2.6 betrachtet werden.

1.1 Notwendigkeit des Prozess-Scheduling

Die Gründe für die Einführung von Schedulingalgorithmen liegen in den Arten von Anwendungen, die auf einem Computersystem ausgeführt werden, begründet. Auf Time-Sharing-Systemen kann eine Vielzahl von Anwendungen laufen. Das Spektrum reicht von Texteditoren über die Verarbeitung wissenschaftlicher Daten bis hin zu multimedialen Anwendungen. Alle diese Aufgaben stellen verschiedene Anforderungen an den Computer und dessen Scheduler. Einige Prozesse fordern nach möglichst viel CPU-Zeit, weil die rechenintensiv sind. Andere Anwendungen benötigen Daten von der Festplatte oder aus Peripheriegeräten. Zudem gibt es noch eine Reihe von Prozessen, die eine Kombination aus den beiden vorhergehenden Anwendungstypen darstellen. Der Scheduler kann sich nicht einmal auf die Konstanz der Anforderungen verlassen, da ein Prozess sein Verhalten im Laufe der Zeit ändern kann.

Oberstes Ziel des Schedulers ist es, möglichst alle Anforderungen zu erfüllen. Ein I/O-intensiver Prozess sollte möglichst schnell seine Daten bekommen, während rechenintensive Prozesse möglichst viel CPU-Zeit erhalten sollten. Dies ist nicht immer möglich. Folglich muss der Scheduler häufig Kompromisse eingehen. Dies geht soweit, dass sich für spezielle Anforderungen verschiedene Schedulingstrategien herauskristallisiert haben.

1.2 Geschichte des Multitaskings auf PCs

Die ersten PCs besaßen keinen Scheduler. Dies lag in der Beschaffenheit der verwendeten Hardware begründet. Der Prozessor war für die Ausführung eines einzelnen Prozesses zur Zeit ausgelegt. Man nahm an, dass der Benutzer nie

mehr als eine Anwendung zur selben Zeit benötigt. Folglich hatten die Betriebssysteme für solche PCs keinen Scheduler. Wenn der Benutzer ein Programm gestartet hat, dann konnte er erst nach dessen Beendigung wieder ein anderes Programm aufrufen.

Im Laufe der Zeit erwies sich diese Vorgehensweise als sehr unpraktisch. Die nächste Stufe in der Entwicklung von Schedulingstrategien war die Einführung des kooperativen Multitaskings (cooperative Multitasking). Dabei handelt es sich um einen Kompromiss zwischen Multitasking und Flexibilität. Die Idee hinter dem kooperativen Multitasking war, wie bereits im Namen erkennbar, die Kooperation der Prozesse untereinander. Diese bestand darin, dass ein Prozess die Kontrolle über die CPU zu gegebener Zeit an das Betriebssystem zurückgeben kann. Das Betriebssystem wählt nun einen anderen Prozess aus und teilt ihm die Kontrolle über die CPU zu. Der Nachteil dieser Vorgehensweise liegt auf der Hand. Es lag einzig und allein am aktuellen Prozess, ob er die CPU freigeben wollte oder nicht. Prinzipiell war es möglich, dass ein Programm die Kontrolle über die CPU überhaupt nicht mehr abgab. Dies kann sich im höchsten Maße negativ auf das System auswirken. Andere Anwendung bekommen dann keine Rechenzeit mehr zugeteilt.

Mit der Entwicklung neuer Hardware konnte ein neues Prinzip des Multitaskings umgesetzt werden – das preemptive Multitasking (preemptive multitasking). Ab sofort war es nicht mehr möglich, dass sich ein Prozess die Kontrolle über die CPU unendlich lange reservieren konnte. Grund dafür war die prinzipielle Unterbrechbarkeit des Prozesses. Nach einer bestimmten, vom Scheduler festgelegten Zeit, wird dem Prozess automatisch die Kontrolle über die CPU entzogen. Der Scheduler wählt nun einen anderen, nach seiner Ansicht besonders geeigneten, Prozess aus und lässt ihn laufen.

Diese Vorgehensweise hat sich bis heute in Betriebssystemen für Personalcomputer nicht geändert.

2. ALLGEMEINE SCHEDULING-STRATEGIEN

In diesem Paper wird eine Klassifizierung der Scheduling-Strategien in die drei Klassen "Scheduling in Batch-Systemen", "Scheduling in interaktiven Systemen" und "Scheduling in Echtzeitsystemen" wie in [1] vorgenommen. Diese drei Klassen sind Gegenstand des nächsten Abschnittes.

2.1 Scheduling in Batch-Systemen

2.1.1 first-come first-served

Auch in Stapelverarbeitungssystemen wurden Schedulingalgorithmen verwendet. Der einfachste unter ihnen ist, wie bereits anfangs angeschnitten, der **first-come-first-served-Algorithmus**. Das Prinzip, das hinter diesem Algorithmus steckt ist einfach. Alle Jobs werden in der Reihenfolge ihres Erscheinens abgearbeitet. Zudem können Prozesse blockieren, wenn Sie auf ein bestimmtes Ereignis (beispielsweise Platten-I/O) warten. Sobald ein Prozess beendet wird oder blockiert, wird der nächste Prozess ausgewählt. Diese Vorgehensweise hat einen entscheidenden Nachteil, weil er rechenintensive Prozesse mit geringem I/O-Aufkommen gegenüber I/O-lastigen Prozessen benachteiligt.

Prozess	A	B	C
Terminiert	Minute 45	Minute 60	Minute 75

Tabelle 1: Terminierungszeiten der Prozesse A, B und C

Prozess	B	C	A
Terminiert	Minute 15	Minute 30	Minute 75

Tabelle 2: Terminierungszeiten der Prozesse mit Shortest-job-first-Algorithmus

2.1.2 Shortest job first

Falls vorher bekannt ist, wie lange jeder Prozess einer Reihe zum Terminieren braucht, eignet sich der Shortest-Job-First-Algorithmus besonders. Wie bereits im Namen erkennbar, wählt dieser Algorithmus immer den Job als nächstes aus, der am kürzesten dauert. Er kann die durchschnittliche Zeit Laufzeit der Prozesse deutlich reduzieren. Dies ist nicht auf den ersten Blick erkennbar. An einem Beispiel wird dies aber sofort klar. Nehmen wir an, es gebe 3 Prozesse A, B und C. A braucht 45 Minuten um zu terminieren, B 15 Minuten und C ebenfalls 15 Minuten. Würde man diese Prozesse in der Reihenfolge ABC ausführen, so ergäben sich für die einzelnen Prozesse Laufzeiten wie in Tabelle 1.

Die Summe l aus den Laufzeiten beträgt dann exakt $45 + 60 + 75 = 180$ Minuten. Die Durchschnittliche Laufzeit δ von n Prozessen mit der Laufzeit z_i beträgt dann

$$\delta = \frac{\sum_{i=0}^n z_i}{n}$$

Minuten – in diesem Beispiel also $\frac{180}{3} = 60$ Minuten.

Bei einer Umstrukturierung der Prozesse aufsteigend nach ihrer Laufzeit ergibt sich die Reihenfolge BCA mit den Laufzeiten wie in Tabelle 2.

Die Durchschnittliche Laufzeit eines Prozesses beträgt nun $\frac{15+30+60}{3} = \frac{105}{3} = 35$ Minuten. Eine Verkürzung der Durchschnittlichen Laufzeit um nahezu 50%.

Wie in [2] bewiesen, ist der Shortest-job-first-Algorithmus aber nur optimal, falls alle Prozesse gleichzeitig lauffähig sind.

2.2 Scheduling in interaktiven Systemen

Wesentlich komplexer als das Scheduling in Batch-Systemen ist das Scheduling in interaktiven Systemen. Neben der allgemeinen Effizienz des Schedulers werden hier noch weitere Anforderungen gestellt. Das System sollte möglichst schnell auf Benutzereingaben reagieren, ansonsten hat der Benutzer den Eindruck eines langsamen und trägen Systems. Im Folgenden werden einige Grundlegende Schedulingalgorithmen aus interaktiven Systemen vorgestellt.

2.2.1 Round-Robin-Scheduling

Analog zum nicht-preemptiven first-come-first-served-Algorithmus in Batch-Systemen existiert das Round-Robin-Verfahren in preemptiven Systemen. Grundlegende Idee ist es, jedem Prozess eine bestimmte Zeitspanne (Quantum) zuzuweisen, in der er laufen darf. Ist dieses Quantum verbraucht, so wird der nächste Prozess ausgewählt (siehe Abb. 1). Ist die Liste der lauffähigen Prozesse durchlaufen, so beginnt der Scheduler wieder beim ersten lauffähigen Prozess. Realisierbar ist solch ein Algorithmus beispielsweise über

eine verkettete Liste, die periodisch vom Scheduler durchlaufen wird. Interessant beim Round-Robin-Scheduling ist

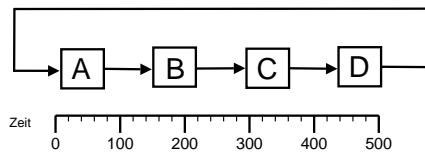


Abbildung 1: Schematischer Ablauf des Round-Robin-Verfahrens

die Wahl der Länge des Quantums. Der Aufruf des Schedulers nach Verbrauch des Quantums eines jeden Prozesses benötigt Zeit. Der Scheduler muss zunächst einen neuen Prozess auswählen und dann einen Prozess-Wechsel (Context-Switch) vornehmen. In der Regel werden dabei die CPU-Register, diverse Pointer (Instruction-Pointer, Stack-Pointer, ...) und die Flagregister auf dem Stack gesichert, um bei erneuter Auswahl dieses Prozesses den Zustand vom Zeitpunkt des Prozesswechsels wieder herstellen zu können. Bei der Wahl von kleinen Quanta wird der Scheduler entsprechend oft aufgerufen. Folglich nimmt der Overhead zu und es wird viel Rechenzeit während der Prozess-Wechsel verbraucht. Zu große Quanta hingegen wirken sich negativ auf die Reaktionszeit des Systems aus. Es gilt, ein Mittelmaß zu finden, das beiden Ansprüchen gerecht wird. Die Wahl von Quanta wird später in diesem Paper noch ausführlicher behandelt (Abschnitt 3: Zeitverhalten).

2.2.2 Prioritätenbasiertes Scheduling

Round-Robin ist in den meisten Fällen nicht optimal. Dies liegt darin begründet, dass verschiedene Prozesse auf einem System häufig auch verschiedene Prioritäten besitzen. Ein Prozess, der im Hintergrund Daten von der Festplatte auf ein Bandlaufwerk sichert hat meistens eine geringere Priorität, als ein Programm, das gerade eine CD brennt und bei zu langer Wartezeit einen Pufferunterlauf riskiert. Ein möglicher Lösungsansatz besteht in der Vergabe von Prioritäten an Prozesse. Dabei haben Prozesse mit einer höheren Priorität eine höhere Wahrscheinlichkeit beim nächsten Aufruf des Schedulers ausgewählt zu werden. Man unterscheidet zudem in dynamische und statische Prioritäten.

Ein gutes Beispiel für die Vergabe von statischen Prioritäten ist die Verwendung des Aufrufs von nice auf UNIX-Systemen. Dem Prozess wird durch entsprechenden Aufruf eine höhere statische Priorität zugewiesen, was bedeutet, dass er gegenüber anderen Prozessen generell bevorzugt wird. Statische Prioritäten sind allerdings nicht die Lösung aller Probleme. Prozesse können ihr Verhalten im Laufe ihrer Lebenszeit verändern. Deshalb macht auch die Vergabe von dynamischen Prioritäten, die erst zur Laufzeit berechnet werden, Sinn. I/O-lastige Prozesse zeichnen sich dadurch aus, dass sie häufig auf Daten aus Ein- und Ausgabe-geräten warten. Sie nutzen in den meisten Fällen also nicht ihr ganzes Quantum aus, sondern geben beim Blockieren durch die Nichtverfügbarkeit bestimmter Daten die Kontrolle automatisch wieder ab. Solche Prozesse sollten durch häufigeres zuweisen von Rechenzeit bevorzugt werden, da hier ein Performancegewinn resultieren kann. Wenn der I/O-lastige Prozess öfters, aber nur kurz läuft, dann kann er sofort die nächste Anfrage nach Daten stellen und blockieren. Während der Bereitstellung der Daten kann dann ein CPU-

Benutzer	1	2	3	4	5	6	7	8	9
% CPU	2	2	2	2	4	4	4	40	40

Tabelle 3: Anteile an der Rechenzeit ohne Fair-Share-Scheduling

lastiger Prozess ausgewählt werden. Somit erreichen beide Prozesse ihre Ziele.

Im Abschnitt über den O(1)-Scheduler des Linux-Kernels ab Version 2.6 wird diese Art des Scheduling genauer betrachtet.

2.2.3 Lottery-Scheduling

Einen anderen Ansatz verfolgt das Lottery-Scheduling. Hier werden jedem Prozess Lose zugeteilt und in bestimmten Abständen eine Verlosung durchgeführt. Die Gewinne bestehen dabei aus CPU-Zeit. So chaotisch, wie der Algorithmus auf den ersten Blick aussehen mag ist er in Wirklichkeit gar nicht. Wenn ein Prozess etwa 50% der Lose besitzt, dann wird er, zumindest über einen längeren Zeitraum gesehen, auch etwa 50% der Rechenzeit in Anspruch nehmen können. Selbst prioritätenbasiertes Scheduling ist hier durch Vergabe von Extra-Losen möglich. Desweiteren können neu gestartete Prozesse relativ schnell zum Zuge kommen, falls ihre Lose entsprechend früh gezogen werden.

2.2.4 Fair-Share-Scheduling

Ein weiteres Problem beim Scheduling tritt auf Mehrbenutzermaschinen auf. Bei gleicher Priorität aller Prozesse auf dem System, unabhängig von deren Besitzern, ist es für einen Benutzer möglich, unfair viel Rechenzeit zu erlangen. Angenommen es gibt 9 Benutzer, die auf einem System eingeloggt sind. Benutzer 1-4 haben jeweils einen Prozess am Laufen, 5, 6 und 7 haben jeweils zwei Prozesse und Benutzer 8 und 9 jeweils 20. Es ergeben sich für die Benutzer die in Tabelle 3 aufgelisteten Anteile an der Rechenzeit. Es ist natürlich sofort ersichtlich, dass Benutzer 8 und 9 wesentlich mehr CPU-Zeit bekommen, als alle anderen Benutzer. Genau hier setzt das Faire-Share-Scheduling-Verfahren an. Im Beispiel könnte die Verteilung der Rechenzeit fairer geschehen, indem von vornherein jedem Benutzer ein gewisser prozentualer Anteil an der CPU-Zeit zugeteilt wird. Bei absoluter Fairness wären das in etwa 11% im oben genannten Beispiel.

2.3 Scheduling in Echtzeitsystemen

Echtzeitsysteme stellen sehr hohe Anforderungen an den Scheduler. Es gibt dort Aufgaben, die innerhalb einer bestimmten Zeit erledigt werden müssen. Je nach System kann ein Nichteinhalten der Fristen zu einer Katastrophe führen. In einem Flugzeug beispielsweise sollte im Falle eines drohenden Zusammenstoßes zweier Flugzeuge möglichst schnell auf Eingaben des Pilots reagiert werden.

Es wird generell zwischen harten und weichen Echtzeitsystemen unterschieden. Harte Echtzeitsysteme haben absolute Fristen für bestimmte Aufgaben, deren Überschreitung ernsthafte Folgen haben könnte. In einem weichen Echtzeitsystem gibt es ebenfalls solche Fristen – eine Überschreitung ist zwar unerwünscht, hat aber nicht unbedingt gravierende Folgen. In diesem Paper werden nur – wenn auch in sehr eingeschränkter Form – weiche Echtzeitsysteme betrachtet.

3. ZEITVERHALTEN

Ein wichtiger Faktor spielt das Zeitverhalten des Schedulers. Wie bereits im Abschnitt über das Round-Robin-Verfahren angesprochen, spielt die Länge eines Zeitquantums eine wichtige Rolle. Mit Zunahme der Geschwindigkeit der Rechner im Laufe der Zeit wird es praktikabler kleinere Zeitquanten zu wählen. Entgegen häufiger Annahmen, dass dies einen übermäßig hohen Overhead erzeugt, zeigten Etsion, Tsafir und Freitelson in [3], dass dies nicht notwendigerweise der Fall ist. Notwendig ist die Wahl kleinerer Zeitquanten allemal, da der Computer nun auch im Bereich der Heimanwender zunehmend für multimediale Datenverarbeitung genutzt wird. Während eine Verzögerung im Bereich von wenigen Millisekunden beim Versand einer E-Mail nicht weiter auffällt, gestaltet sich dies beim Abspielen von Klängen und Musik schon ganz anders. Wird der Puffer zur Audioausgabe nicht rechtzeitig gefüllt, dann nimmt der Benutzer dies schon bei minimalen Verzögerungen als Knacken, Rauschen oder Aussetzen wahr. Ähnlich verhält es sich bei der Wiedergabe von Videos. Eine Unregelmäßigkeit bei der Darstellung einzelner Frames führt zu deutlich erkennbarem Haken und Springen.

Umso interessanter sind die Ergebnisse aus [3]. Sie basieren im Wesentlichen darauf, dass die Anzahl der Takte, die ein Context-Switch benötigt in etwa gleich geblieben ist. Mit höherer Taktrate sinkt somit die Zeit, die für einen Prozesswechsel benötigt wird. Genau dies ermöglicht die Wahl kleinerer Quanta. Inzwischen nutzt Linux ab Version 2.6 in der Standardeinstellung zehnmals so kleine Quanta wie zuvor: 1000Hz statt 100Hz (bezogen auf die weit verbreiteten IA32-Prozessoren).

4. REAKTIONSZEITEN

Natürlich bringt eine höhere Auflösung der Quanta auch bessere Antwortzeiten mit sich. Dies gilt allerdings hauptsächlich für traditionelle Systeme, in denen klar zwischen Hintergrund-Jobs mit hoher CPU-Last und Vordergrundprozessen mit hohem I/O-Aufkommen unterschieden werden kann. Wie bereits eingangs erwähnt, hat sich das Verhalten der Nutzer aber besonders in den letzten Jahren geändert. So gilt ein Prozess, der für das Abspielen eines komprimierten Videos verantwortlich ist, nach dem alten Schema nicht unbedingt als interaktiv. Die Menge an Daten, die der Prozess von Ein- und Ausgabegeräten benötigt ist nicht sehr groß. Viel mehr Zeit nimmt das Dekodieren des Videos in Anspruch. Der Prozess gilt folglich eher als CPU-lastig und somit als nicht -interaktiv. Solange nur dieser Prozess auf dem Rechner läuft, erzeugt dies keine weiteren Probleme. Bei zunehmender Anzahl von rechenintensiven Hintergrundprozessen erfolgt aber eine schlechtere Verteilung der CPU-Zeit, da alle Prozesse als Hintergrundjobs betrachtet werden. Etsion, Tsafir und Freitelson fanden in [4] einen möglichen Lösungsansatz für dieses Problem. Eine gängige Praxis war das Verschieben der Schedulingmechanismen in den Userspace – der Nutzer war gezwungen entweder selbst auf den Workload des Rechners zu achten, oder mittels geeigneter Tools (nice, etc.) die statische Priorität der Prozesse zu regeln. Dies ist, wie in [4] ebenfalls aufgezeigt, kein besonders guter Ansatz. Ein anderer Ansatz war die Einführung komplexerer APIs, um Fehler bei der Erkennung interaktiver Prozesse auszugleichen. Dies führt zu einer schlechteren Portabilität der Software und zu einem erhöhten Lern- und

Codeumfang.

Etsion, Tsafir und Freitelson weisen auf ein ganz spezielles menschliches Merkmal hin: Der Mensch bringt am meisten Aufmerksamkeit für Bewegungen auf. Desweiteren untersuchten sie das Verhalten des Benutzers in Bezug auf Prozesse, die Bewegungen visuell auf dem Bildschirm darstellen. Bewegungen auf dem Bildschirm, die mit der aktuell vom Benutzer verfolgten Aufgabe nichts zutun haben, wirken störend und lenken ab. Der Nutzer kümmert sich folglich, ob bewusst oder unbewusst, selbst darum, dass möglichst wenige störende Bewegungsquellen auf dem Bildschirm sichtbar sind. Dies kann in einem traditionellen grafischen System beispielsweise durch Minimierung oder Ausblendung von Fenstern geschehen. Was übrig bleibt sind die Bewegungsquellen, die für den Benutzer interessant sind.

Bewegung auf dem Bildschirm unterscheidet sich grundlegend von der in der realen Welt. Es handelt sich weniger um Gegenstände, die sich bewegen, sondern vielmehr um einzelne oder Gruppen von Pixeln, die ihren Zustand ändern. Der in [4] grob beschriebene Algorithmus setzt genau hier ein. Gemessen werden die Art und Anzahl von Grafikoperationen einer Applikation. Je höher diese ist, desto mehr Bewegung herrscht und desto eher handelt es sich um einen für den Benutzer interessanten Prozess. Die erzielten Ergebnisse sind durchaus bemerkenswert. Trotz hoher Anzahl an CPU-lastigen, nichtvisuellen Prozessen bleibt gleichzeitig die Framerate eines Videoplayers mit ebenfalls hoher CPU-Last durchaus akzeptabel.

Der Ansatz lässt nichtvisuelle Prozesse, die trotzdem für den Benutzer interessant sein können außer Acht (beispielsweise MP3-Player). Es ist allerdings nicht auszuschließen, dass sich dieses Konzept auch in diese Richtung ausweiten lässt.

5. ABSCHLUSS

Auch Jahrzehnte nach der Entwicklung der ersten Schedulingalgorithmen gibt es immernoch offene Fragen. Dadurch, dass sich die Anwendungsgebiete des Computers von einfachen Aufgaben in Richtung Multimedia und verstärkter Interaktivität verschoben haben, sind neue Probleme aufgetreten. Teilweise konnten die Probleme in jüngerer Zeit gelöst werden. Es bleibt abzuwarten, was für Anforderungen an zukünftige Schedulingalgorithmen gestellt werden. Allgemeine Lösungen mit Bezug zu aktueller Problematik werden nicht für alle Ewigkeit von Bestand bleiben. Mit der Zeit verändern sich Betriebssysteme und als Teil von ihnen auch die Scheduler sowie deren Algorithmen.

ANHANG

A. DER LINUX-SCHEDULER

Im Folgenden soll der $O(1)$ -Scheduler des Linux-Kernels¹ betrachtet werden. Die Angaben beziehen sich allesamt auf die Kernel-Versionen ab Version 2.6. Gute Einführungen in den Kernel 2.6 bieten [5] und [6]. Im Laufe der Entwicklung von Kernel 2.6 wurde der Scheduler größtenteils neu geschrieben. Ein wichtiges Ziel bei der Entwicklung des neuen Algorithmus war, dass der Scheduler eine konstante Zeit für Prozesswechsel benötigt – unabhängig von der Anzahl der Prozesse im System. In diesem Ziel liegt auch der Ursprung des Namens des $O(1)$ -Schedulers begründet. $O(1)$ steht für die Komplexität 1, also für eine konstante, invariable Zeitspanne der Ausführung, unabhängig von externen Einflüssen.

A.1 Schedulingstrategie

Wie alle derzeit verfügbaren und universellen Multitasking-Betriebssysteme verwendet auch Linux preemptives Multitasking in Verbindung mit prioritätsbasiertem Scheduling. Prozesse mit einer höheren Priorität erhalten längere Zeitscheiben (time-slices). Time-Slices beschreiben die Zeit, die ein Prozess im Durchlauf des Schedulers laufen darf, sind also synonym zum Begriff des Zeitquantums. Wichtig ist in diesem Zusammenhang, dass die Time-Slice nicht in einem Zug aufgebraucht werden muss. Es ist möglich, dass in einem Durchlauf des Schedulers ein Prozess stückchenweise seine Time-Slice verbraucht.

Der Linux-Kernel baut auf der Idee des dynamic priority-based scheduling auf, bei der es neben statischen auch dynamische Prioritäten gibt. Die statischen Prioritäten können über den Aufruf von nice beeinflusst werden. Somit ist sichergestellt, dass der Benutzer Einfluss auf das Verhalten des Schedulers nehmen kann. Die Priorität definiert gleichzeitig die Länge der Time-Slice. Die Länge der Quanta ist also zwischen den Prozessen variabel. Natürlich muss es einen Bezug zu einem Standardquantum geben, damit die Quanta der verschiedenen Prioritäten berechnet werden können. Viele Betriebssysteme wählen kleine Standardquanta, um schnelle Prozesswechsel zu erzwingen und somit die Interaktivität des Systems zu steigern. Im Linux-Kernel werden standardmäßig relativ lange Quanta von 100ms benutzt. Im Gegensatz zu anderen Betriebssystemen führt dies aber nicht zu geringerer Interaktivität. Dies liegt in zwei bestimmten Eigenschaften des Schedulers begründet:

1. Der Scheduler wählt im Allgemeinen den Prozess aus, der die höchste Priorität besitzt und die längste Time-Slice besitzt.
2. Quanta müssen nicht in einem Zug verbraucht werden (Prinzip der Reschedules).

Die minimale Time-Slice beträgt 5ms und das absolute Maximum 800ms.

Die dynamische Priorität wird als Realtime-Priorität (Echtzeitpriorität) bezeichnet. Sie wird, zumindest teilweise, erst zur Laufzeit bestimmt. Interaktive Prozesse erhalten automatisch eine höhere dynamische Priorität. Linux betrachtet I/O-lastige Prozesse als interaktiv. Trotz der Probleme, die

¹Auf <http://www.kernel.org/> erhältlich

bei dieser Annahme auftreten können, erzielt Linux recht gute Ergebnisse in Bezug auf die Interaktivität. Dies liegt teilweise auch darin begründet, dass durch die Implementierung spezieller Mechanismen selbst der Kernel an sich an gewissen Stellen unterbrechbar ist (preemptible Kernel).

Der Linux-Scheduler ist nicht nur auf Einprozessormaschinen aus dem Desktop-Bereich einsetzbar. Er wurde entwickelt, um eine Vielzahl von Situationen abzudecken und selbst auf hochparallelierten Maschinen mit weit mehr als nur zwei Prozessoren laufen zu können. Sogar eine Lastverteilung zwischen den einzigen Prozessoren wird zuverlässig erledigt. Wie dies im Einzelnen geschieht sei später erklärt.

A.2 Scheduler-Aufrufe

Der Scheduler wird im Wesentlichen in zwei Situationen aufgerufen. Zum einen, wenn ein Prozess seine Time-Slice aufgebraucht hat. Dies ist notwendig, damit auch die anderen Prozesse im selben Scheduling-Zyklus an die Reihe kommen. Zum anderen wird der Scheduler immer dann aufgerufen, wenn ein Prozess mit höherer Priorität als der des aktuell laufenden Prozesses lauffähig wird. Somit bleibt gewährleistet, dass Prozesse mit höherer Priorität vom Scheduler bevorzugt werden.

A.3 Runqueues

Jeder Prozessor besitzt eine Runqueue, die aus zwei Priority-Arrays besteht. Ein Priority-Array ist eine Struktur, die neben Prozessinformationen auch solche über die Anzahl der lauffähigen Tasks und Prioritätslevel enthält. Sie ist wie folgt implementiert:

```
struct prio_array {
    int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};
```

nr_active enthält die Anzahl lauffähiger Prozesse in diesem Array. Das Array queue besteht aus MAX_PRIO Listenköpfen für Prozesswarteschlangen. Üblicherweise ist MAX_PRIO mit dem Wert 140 belegt. Genau dies ist die Anzahl an verschiedenen Prioritätsstufen für Prozesse. Innerhalb des queue-Arrays tauchen also die Prozesse geordnet nach ihrer Priorität auf. Ein weiteres Element der Struktur ist das Array bitmap, welches ein wichtiger Bestandteil der Linux-Schedulingstrategie ist. Jedes Bit der Bitmap repräsentiert das Vorhanden- oder Nichtvorhandensein eines lauffähigen Prozesses in der jeweiligen Prioritätsstufe.

Angenommen es gibt in der Warteschlange für Prozesse mit der Priorität 10 ein oder mehrere Prozesse die lauffähig sind. In diesem Fall ist das zehnte Bit in bitmap auf 1 gesetzt. Somit lassen sich in konstanter Zeit lauffähige Prozesse finden, die beim Scheduling eingeplant werden müssen. Zudem sind Bitoperationen im Vergleich zum Durchsuchen einer Liste oder eines Arrays generell sehr schnell. Die meisten von Linux unterstützten Architekturen bieten eine spezielle Instruktion, mit der sich das erste gesetzte Bit in einem Wort oder einer Reihe von Worten finden lässt. Ist ein Bit in der für das Scheduling aktuell ausgewählten Prioritätsstufe gesetzt, so wird begonnen die Warteschlange dieser Priorität nach dem Round-Robin-Verfahren abzuarbeiten.

Wie bereits erwähnt besteht jede Runqueue aus zwei Priority-Arrays. Das erste der beiden Arrays enthält die akti-

ven, laufbereiten Prozesse. Das zweite Array hingegen enthält genau die Prozesse, deren Time-Slice aufgebraucht wurde. Das Verschieben von Prozessen vom aktiven Array in das abgelaufene Array erfolgt direkt nachdem seine Time-Slice aufgebraucht ist. Auch dieser Punkt ist besonders wichtig in Bezug auf die Scheduling-Strategie, die der Linux-Kernel verfolgt. Da Prozesse ihre Interaktivität im Laufe der Zeit verändern können, muss laufend die Priorität neu bestimmt werden. Dies geschieht beim Verschieben in die Liste der abgelaufenen Prozesse. Es wäre äußerst ungünstig dies zu einem anderen Zeitpunkt zu tun, denn nur so ist gewährleistet, dass der Aufruf des Schedulers eine konstante Zeit benötigt. Es könnte sonst sein, dass die Priorität mehrerer Prozesse neu bestimmt werden muss. Und dies wäre inakzeptabel in Hinblick auf die Echtzeitfähigkeit des Schedulers. Was passiert nun aber, wenn alle Prozesse ins inaktive Array verschoben wurden? Anscheinend ist dies genau der Punkt, an dem ein Scheduling-Zyklus vollendet wurde. An dieser Stelle werden die Pointer vom aktiven und inaktiven Array vertauscht und es beginnt ein neuer Scheduling-Zyklus. Außer dem Vertauschen ist keine weitere Arbeit nötig.

A.4 Echtzeitfähigkeit

Der $O(1)$ -Scheduler hat den Anspruch auch weichen Echtzeitprozessen eine zufriedenstellende Umgebung zu bieten. Harte Echtzeitfähigkeit wird von Linux nur in speziell modifizierten Versionen geboten. Diese gehören nicht zum Mainstream-Kernel und stellen separate Entwicklungszweige dar.

A.5 Lastverteilung auf Mehrprozessormaschinen

Da Linux seit Version 2 auch auf SMP-Systemen² lauffähig ist, muss auch der Scheduler besondere Fähigkeiten aufweisen. Eine davon ist die Lastverteilung (load-balancing) auf die im System vorhandenen Prozessoren. Es wäre ineffektiv in einem Zweiprozessorsystem 90% aller Prozesse auf der ersten und 10% aller Prozesse auf der zweiten CPU laufen zu lassen. Der Scheduler verteilt die Prozesse gleichmäßig auf die vorhandenen CPUs. Dies geschieht an zwei Stellen im System. Wie bereits erwähnt besitzt jeder Prozessor seine eigene Runqueue. Ist die Runqueue einer CPU leer, so wird der Lastverteilungsmechanismus ausgeführt. Natürlich führt dies nicht zu einer gleichmäßigen Verteilung. Für deren Gewährleistung wird die Lastverteilung periodisch alle 200ms durchgeführt.

2. REFERENZEN

- [1] Tanenbaum, Andrew S. *Moderne Betriebssysteme. 2., überarbeitete Auflage*, 2002. Pearson Studium / Prentice Hall.
- [2] Tanenbaum, Andrew S. *Moderne Betriebssysteme. 2., überarbeitete Auflage, Kapitel 2, S. 157*, 2002. Pearson Studium / Prentice Hall.
- [3] Etsion, Yoav; Tsafir, Dan; Freitelson, Dror G. *Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-time Processes*. In *SIGMETRICS '03* S. 172 - 183.
- [4] Etsion, Yoav; Tsafir, Dan; Freitelson, Dror G. *Desktop Scheduling: How Can We Know What the User Wants?*. In *NOSSDAV '04* S. 110 - 115.

²Symmetric multiprocessing

- [5] Love, Robert. *Linux-Kernel-Handbuch - Leitfaden zu Design und Implementierung von Kernel 2.6*, 2005. Addison-Wesley.
- [6] Mauerer, Wolfgang. *Linux Kernelarchitektur - Konzepte, Strukturen und Algorithmen von Kernel 2.6*, 2004. Carl Hanser Verlag München Wien.