Prof. Dr. Kelter, Prof. Dr. Klein, Dr. Lihong Ma,
apl. Prof. Dr. Christian Icking

Modul 63212

Betriebssysteme

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhalt

1 Einführung **5**

2 Prozesse **49**
2.1 Programme und Prozesse . . . 49
2.2 Prozessmerkmale . . . 50
2.2.1 Prozesszustände . . . 51
2.2.2 Speicherbereich . . . 51
2.2.3 Prozesskontrollblock . . . 52
2.2.4 Prozesshierarchien . . . 53
2.3 Zustandsübergänge . . . 53
2.4 Prozesswechsel . . . 55
2.5 Scheduling . . . 57
2.5.1 Qualitätsmaßstäbe von Scheduling-Strategien . . . 58
2.5.2 Scheduling für nicht-präemptive Systeme . . . 60
2.5.2.1 First Come First Served (FCFS) . . . 60
2.5.2.2 Shortest Job First (SJF) . . . 62
2.5.2.3 Priority Scheduling . . . 64
2.5.3 Scheduling für präemptive Systeme . . . 65
2.5.3.1 Round Robin . . . 65
2.5.3.2 Shortest Remaining Time First . . . 68
2.5.3.3 Priority Scheduling . . . 68
2.5.4 Kombinationen der Scheduling-Strategien . . . 69
2.5.4.1 Feedback Scheduling . . . 69
2.5.4.2 Mehrere Warteschlangen . . . 70
2.5.4.3 Scheduling in Linux . . . 72
2.5.5 Auswahl einer Scheduling-Strategie . . . 73
2.6 Vorteile und Probleme des allgemeinen Prozessmodells . . . 74
2.7 Leichtgewichtige Prozesse . . . 76
2.7.1 Realisierungen von Threads . . . 77
2.7.2 Anwendungsgebiete für Threads . . . 80
2.8 Zusammenfassung . . . 81
Literatur . . . 83
Glossar . . . 89

3 Hauptspeicherverwaltung **95**

4 Prozesskommunikation **151**

5 Geräteverwaltung und Dateisysteme **203**

6 Sicherheit **259**

7 Kommandosprachen **315**

Kurseinheit 2

Prozesse

2.1 Programme und Prozesse

Wie schon in Kurseinheit 1 erklärt wurde, sind schnelle Prozessoren mit der Abarbeitung eines einzigen Programms in der Regel unterfordert. Wir gehen daher im Folgenden davon aus, dass der Rechner im Mehrprogrammbetrieb (*Multiprogramming*, *Multitasking*) betrieben wird. Während ein Programm z. B. auf Eingaben des Benutzers wartet, kann der Prozessor mit der Bearbeitung eines anderen Programms weitermachen. Oder der Prozessor kann regelmäßig jeweils ein Stück eines Programms bearbeiten und dann mit einem anderen Programm weitermachen. Man sagt, dass der Prozessor zwischen verschiedenen Programmen hin- und hergeschaltet wird.

Mehrprogramm-betrieb

Es sollen also mehrere Programme quasi gleichzeitig (als *parallele Prozesse*) abgearbeitet werden. Die Kontrolle dieser parallelen Aktivitäten ist nicht einfach. In dieser Kurseinheit wollen wir uns damit befassen, wie das Modell der Prozesse hilft, diese Parallelität in Systemen einfacher zu beschreiben und zu beherrschen.

Quasiparallelität

Außerdem werden wir vorstellen, wie es möglich ist, dass der Prozessor zwischen zwei Prozessen hin- und hergeschaltet werden kann. Dann stellen wir eine Reihe von Strategien vor, wie das Betriebssystem aus der großen Zahl von bereiten Prozessen einen Prozess auswählen kann, der als nächster den Prozessor zugeteilt bekommt. Die Auswahl einer solchen Strategie bestimmt das Verhalten des Rechnersystems in entscheidendem Maße.

Scheduling-Strategie

Es ist ausgesprochen schwierig, den Begriff Prozess so allgemein zu definieren, dass er auf alle Rechner und Betriebssysteme sinnvoll anwendbar ist. Wir werden daher keine formale Definition angeben, sondern wollen den Begriff informell beschreiben.

Aus dem bisher Gesagten ist schon klar, dass die Begriffe Programm und Prozess irgendwie zusammengehören. Wir wollen uns deshalb zunächst kurz mit dem Begriff Programm genauer befassen. Dann werden wir die Unterschiede zwischen einem Programm und einem Prozess aufzeigen.

Programme. Programme können in sich parallel oder sequentiell sein. Wir beschränken uns auf die Klasse der sequentiellen Programme, wie sie in den

meisten gängigen Programmiersprachen formuliert werden können. Sequentiell heißt in diesem Zusammenhang, dass während der Ausführung des Programms zu jedem Zeitpunkt höchstens eine Anweisung des Programms ausgeführt wird.

Programm
Algorithmus

Das Programm ist eine Formulierung eines Algorithmus, d. h. es beschreibt in endlicher Form, welche Operationen in welcher Reihenfolge ausgeführt werden sollen. Der Programmtext ist die vollständige Beschreibung des Programms.

Prozess. Wird ein Programm von einem Prozessor abgearbeitet, so reicht der Programmtext zur vollständigen Beschreibung nicht mehr aus. Der Prozessor hat bereits eine Reihe von Operationen ausgeführt. Er kennt die nächste auszuführende Operation (Programmzähler), und seine Register enthalten berechnete Daten. Während der Programmtext normalerweise auf sekundären Speichern (Festplatten, Bänder) steht, befindet sich ein Programm zur Ausführung im Hauptspeicher. Falls das Programm Dateien bearbeitet, so hat es diese evtl. geöffnet und schon teilweise gelesen. Damit können wir jetzt den Begriff Prozess wie folgt beschreiben:

Programmzähler
Registerinhalt

Prozess

Ein ablaufendes Programm bezeichnet man als **Prozess**, inklusive des aktuellen Werts des Befehlszählers, der Registerinhalte und der Belegungen der Variablen.

Unterschiede. Wichtig an dieser Festlegung ist der kleine Unterschied zwischen den Begriffen Programm und Prozess. Ein Programm an sich ist etwas statisches, während ein Prozess etwas dynamisches ist. Tanenbaum [Ta02] hat dies durch eine Analogie sehr schön gezeigt. Ein Mensch will einen Kuchen backen. Er hat ein Kochbuch, in dem das Rezept für die Zubereitung steht. Außerdem hat er alle Zutaten. Das Rezept entspricht in dieser Analogie dem *Programm*, der Mensch entspricht dem *Prozessor*, die Zutaten den Eingabedaten und der Vorgang des Kuchenbackens entspricht dem *Prozess*.

Es ist anhand dieser Analogie auch leicht einzusehen, dass mehrere Prozesse dasselbe Programm ausführen können (Mehrere Menschen backen nach demselben Rezept). Trotzdem sind diese Prozesse natürlich nicht gleich, da sie z. B. verschieden weit in ihrer Ausführung vorangekommen sein können. Außerdem sieht man leicht ein, dass ein Prozessor (Mensch) mehreren Prozessen (Apfelkuchen backen, Zeitung lesen, spülen) abwechselnd zugeteilt werden kann.[1]

2.2 Prozessmerkmale

Prozesstabelle

In einem laufenden Betriebssystem gibt es immer sehr viele Prozesse gleichzeitig, diese müssen in einer so genannten **Prozesstabelle** verwaltet werden. Was die einzelnen Merkmale sind, die das Betriebssystem über jeden Prozess wissen muss, wollen wir in diesem Abschnitt präzisieren.

[1] Wer oder was in dieser Analogie dann das Betriebssystem ist, das die Zuordnung übernimmt, bleibt der Phantasie der Leserinnen und Leser überlassen.

2.2.1 Prozesszustände

Jeder Prozessor eines Rechners (oft gibt es nur einen einzigen, immer häufiger aber auch zwei oder gar vier) kann zu einem bestimmten Zeitpunkt nur einen Prozess abarbeiten. Weitere im System vorhandene Prozesse müssen warten. Dabei unterscheidet man Prozesse, die nur darauf warten, dass ein Prozessor frei wird, und Prozesse, die auf Ein-/Ausgaben warten. Letztere können in ihrer Ausführung ja selbst dann nicht fortfahren, wenn sie einen Prozessor zugeteilt bekämen. Wir unterscheiden also folgende **Prozesszustände**:

Prozesszustände

erzeugt: Der Prozess ist gerade neu entstanden, d. h. das Betriebssystem erstellt die Datenstrukturen, die für die Verwaltung des Prozesses verwendet werden, und weist ihm Adressraum im Hauptspeicher zu.

bereit: Der Prozess ist rechenbereit, hat aber noch keinen Prozessor zugeteilt bekommen. Er wartet nur darauf, dass ein Prozessor für ihn frei wird.

rechnend: Ein Prozessor ist dem Prozess zugeteilt und arbeitet die Anweisungen des zugehörigen Programms ab.

blockiert: Der Prozess wartet auf irgendein Ereignis, z. B. Ein-/Ausgabe.

beendet: Das Ende der Ausführung des zugehörigen Programms ist erreicht.

Im Zustand *rechnend* kann nur ein Prozess pro vorhandenem Prozessor sein. Dagegen können viele Prozesse in den Zuständen *erzeugt*, *bereit*, *blockiert* und *beendet* sein. Diese Prozesse werden dann in Listen verwaltet.

2.2.2 Speicherbereich

Wie schon gesagt, muss ein Prozess im Hauptspeicher des Rechners sein, damit er vom Prozessor ausgeführt werden kann. Wie das genau realisiert wird, ist Thema der Kurseinheit 3 (Hauptspeicherverwaltung). Im Moment gehen wir einfach davon aus, dass das Betriebssystem dem Prozess (wie auch immer) einen *logischen Speicherbereich* zuteilt. Man spricht in diesem Fall auch vom *logischem Adressraum* des Prozesses. In diesem Bereich kann der Prozessor den Prozess ausführen.

Kennzeichnend für einen Prozess ist nun, womit dieser seinen Speicherbereich belegt. Der logische Speicherbereich eines Prozesses enthält:

Programmsegment Es enthält den vom Prozessor ausführbaren Code des Programms.

Programmsegment

Stacksegment Hier steht der Programmstack, auf den bei jedem Prozeduraufruf ein Aktivierungsblock mit den Parametern, lokalen Variablen und der Rücksprungadresse kommt.

Stacksegment

Datensegment Hier stehen die Daten, die das Programm bearbeiten soll oder die es erzeugt hat.

Datensegment

Während der Abarbeitung des Prozesses ändert sich das Programmsegment nicht[2], jedoch ändern sich Stack- und Datensegment mit dem Fortschritt der

[2] Von Programmen, die selbstmodifizierenden Code enthalten, sehen wir hier einmal ab.

Abarbeitung. Im Moment ist für unsere Betrachtungen allerdings nur wichtig, dass

1. das Betriebssystem dem Prozess zu seinem Beginn einen logischen Speicherbereich zur Verfügung stellt.
2. nach dem Ende des Prozesses dieser Speicherbereich wieder freigegeben wird.

Was der Prozess zwischen seiner Entstehung und seinem Verschwinden in seinem Speicherbereich macht, ist in diesem Kapitel nicht weiter wichtig.

2.2.3 Prozesskontrollblock

Prozesskontrollblock

Alle Daten, die das Betriebssystem über einen Prozess verwalten muss, bezeichnet man als **Prozesskontrollblock** oder kurz **PCB** (engl. **process control block**), und alle PCBs zusammen werden in einer **Prozesstabelle** organisiert. Im Prozesskontrollblock stehen insbesondere folgende Informationen.

Prozessnummer

Prozessidentifikation. Jeder Prozess muss eindeutig identifizierbar sein, üblicherweise durch eine **Prozessnummer** (**Prozess-ID**). Diese Nummer kann als Index benutzt werden, um einen Prozess in der Prozesstabelle zu lokalisieren.

Programmzähler
Register

Prozessorstatus. Wenn ein Prozess wieder in den Zustand *rechnend* übergehen soll, so muss der Prozess die Abarbeitung seines Programms an genau der Stelle fortsetzen, an der er vorher stand. Das Betriebssystem muss also den **Programmzähler**, der die Adresse des nächsten auszuführenden Befehls enthält, wieder restaurieren. Außerdem müssen alle anderen **Register** wieder ihre alten Werte erhalten.

Prozesskontrollinformationen. Darunter versteht man u. a. folgende Angaben.

Prozesszustand
Priorität

- Der aktuelle **Prozesszustand** und die **Priorität** des Prozesses für das Scheduling.

Speicherbereich

- Informationen über den **Speicherbereich**, in dem der Prozess abläuft. Das Betriebssystem muss unterschiedliche Prozesse voreinander schützen, also Speicherzugriffe eines Prozesses in den Speicherbereich eines anderen Prozesses erkennen und verhindern (*Grenzregister*).

geöffnete Dateien

- Informationen über alle **geöffneten Dateien** des Prozesses.

Buchhaltung

- Buchhaltung: Neben den bisher genannten Daten verwaltet das Betriebssystem weitere Informationen. Zum Beispiel wird auch darüber Buch geführt, wie lange ein Prozess bisher gerechnet hat, wieviel Hauptspeicher er belegt usw. Diese Informationen kann man z. B. für das Scheduling benutzen, und auch dazu, den Benutzern ihre verbrauchten Ressourcen in Rechnung zu stellen. Natürlich muss bei jedem Prozess vermerkt sein, welchem Benutzer er zugeordnet ist.

Ressourcenverbrauch

Besitzer

Diese Aufzählung ist nicht unbedingt vollständig. Je nach Betriebssystem stehen im Prozesskontrollblock noch weitere oder aber auch weniger Informationen.

Das Programm, der Stack, die Daten und der Prozesskontrollblock eines Prozesses werden zusammen als **Prozessabbild** bezeichnet.

Prozessabbild

2.2.4 Prozesshierarchien

Ein Prozess kann wieder Kindprozesse erzeugen, aus einem Prozess und seinen Nachkommen entsteht eine Prozessfamilie, die eine Prozesshierarchie bildet. In UNIX gibt es zwei spezielle Prozesse mit den Nummern 0 und 1. Wenn das System gebootet wird, wird *Prozess 0* erzeugt, der die Echtzeituhr einrichtet, das erste Dateisystem verfügbar macht und einen *Prozess 1* erzeugt, der auch als *init-Prozess* bezeichnet wird. Alle weiteren Prozesse im System stammen von *init* ab insbesondere die Prozesse, die für das Funktionieren des gesamten Systems im Hintergrund wichtige Aufgaben verrichten. Der Prozess *init* erzeugt auch mehrere *login-Prozesse*, die warten, bis sich jemand anmeldet. Nachdem sich ein Benutzer korrekt eingeloggt hat, startet der *login*-Prozess weitere Prozesse, die zusammen die Benutzerumgebung bilden. Diese können wiederum Prozesse starten, siehe Abbildung 2.1. Somit gehören alle Prozesse im gesamten System zu einem einzigen Baum mit *Prozess 0* an seiner Wurzel. Unter Windows hingegen gibt es kein Konzept einer Prozesshierarchie, es wird keine Beziehung zwischen Vater- und Kindprozessen erzwungen, alle Prozesse sind gleichwertig.

init-Prozess

login-Prozesse

In UNIX verwendet man die Kommandos *ps*, *pstree* und *top*, um die laufenden Prozesse und ihren Ressourcenverbrauch anzuzeigen. Unter Windows 95/98/ME/2000/XP kann man die Tastenkombination CTRL-ALT-DEL benutzen, um den Taskmanager aufzurufen, der auch die laufenden Prozesse anzeigt.

2.3 Zustandsübergänge

Zwischen den Prozesszuständen sind nun verschiedene Übergänge möglich. In Abbildung 2.2 sind noch einmal die fünf Zustände und die möglichen Übergänge in einem *Zustandsübergangsdiagramm* dargestellt.

Zustands-übergangs-diagramm

Wenn ein neuer Prozess erzeugt wird, so tritt dieser neue Prozess kurz in den Zustand *erzeugt* ein (1). Damit das zugehörige Programm tatsächlich ausgeführt werden kann, müssen dem Prozess einige Ressourcen zugeteilt werden. Die wichtigste davon ist der Hauptspeicher, in dem der Prozess ablaufen soll. Hat das System einen logischen Speicherbereich für den Prozess festgelegt, so geht der Prozess in den Zustand *bereit* über (2). Der Prozess wartet nun darauf, dass er den Prozessor zugeteilt bekommt, damit er abgearbeitet werden kann.

erzeugt

bereit

Nachdem der bereite Prozess den Prozessor zugeteilt bekommt, geht er in den Zustand *rechnend* über (3). In diesem Zustand bleibt der Prozess eine Weile. Wartet der Prozess dann auf ein Ereignis (z. B. Ein-/Ausgaben), so

rechnend

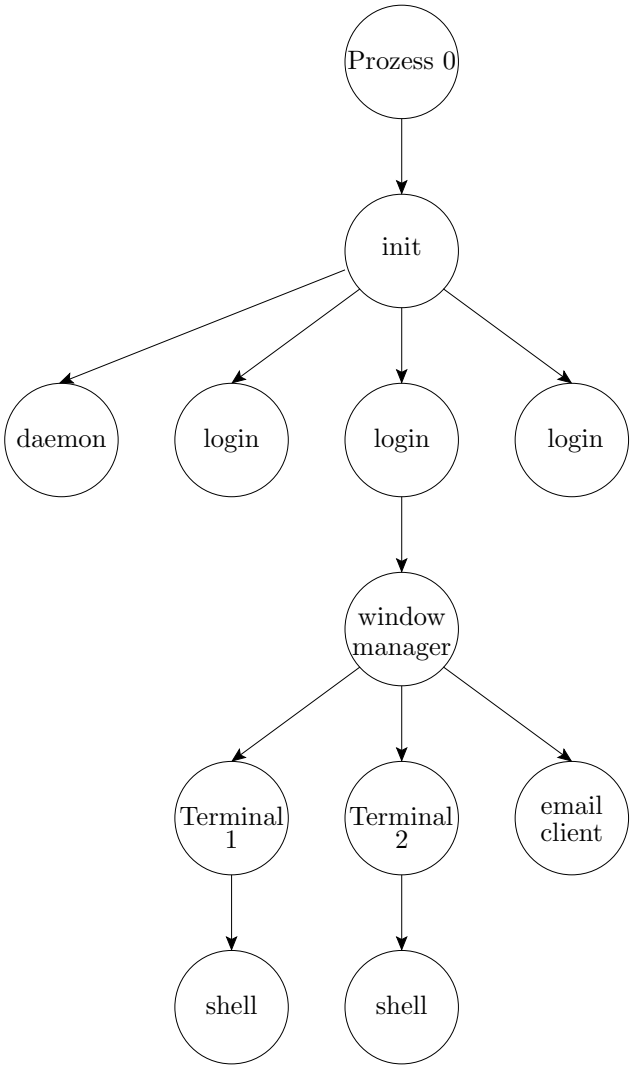

Abbildung 2.1: UNIX/Linux-Prozessbaum.

blockiert

geht er in den Zustand *blockiert* (5). Für jeden Ereignistyp existiert eine eigene Liste (Warteschlange), in der die Prozesse einsortiert werden, die auf ein Ereignis dieses Typs warten. Mögliche Ereignistypen sind: Tastendruck, Mausklick, Plattenzugriff, Bereitmeldung vom Drucker, Nachrichten von einem anderen Rechner eines Rechnernetzes usw. In Abbildung 2.3 sind die einzelnen Warteschlangen dargestellt.

In diesem Zustand bleibt der Prozess, bis das Ereignis eingetreten ist. Wenn das Ereignis eintritt, wird in der zugehörigen Liste der blockierten Prozesse ein Prozess, der auf dieses Ereignis wartet, herausgesucht. Dieser Prozess geht in den Zustand *bereit* über (6).

Übungsaufgabe 2.1 Erklären Sie, warum kein direkter Übergang vom Zustand *blockiert* in den Zustand *rechnend* vorgesehen ist.

Der direkte Übergang vom Zustand *rechnend* in den Zustand *bereit* (4) kann zwei Ursachen haben. Zum einen kann ein Prozess von sich aus entscheiden, dass er den Prozessor an einen anderen Prozess abgeben will. Dazu ruft

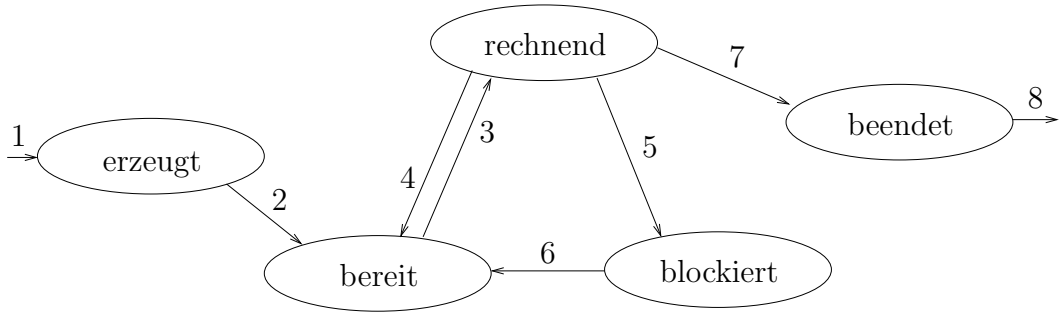

Abbildung 2.2: Prozesszustandsübergangsdiagramm.

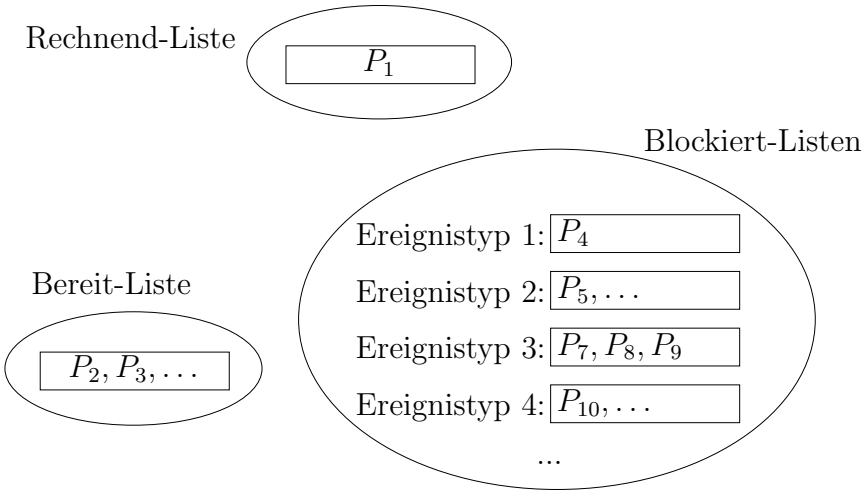

Abbildung 2.3: Listen von Prozessen in den jeweiligen Zuständen.

er eine Systemfunktion auf, das Betriebssystem erhält die Kontrolle und kann einen anderen Prozess in den Zustand *rechnend* versetzen. Betriebssysteme, bei denen nur der Prozess selbst entscheidet, ob er den Prozessor abgeben will, nennt man **nicht-präemptiv** (engl. **non-preemptive**). Die zweite Möglichkeit ist, dass das Betriebssystem einem laufenden Prozess den Prozessor entzieht. Wie das genau vor sich geht, wird später erklärt. Solche Systeme nennt man **präemptiv** (engl. **preemptive**).

nicht-präemptiv

präemptiv

Terminiert der Prozess, so geht er in den Zustand *beendet* (7). Der von ihm belegte Speicherbereich sowie weitere vom Prozess gebundene Ressourcen werden freigegeben. Dann verschwindet der Prozess aus dem System (8).

2.4 Prozesswechsel

Wenn das Betriebssystem die Kontrolle vom gerade rechnenden Prozess erhält, dann erfolgt ein **Prozesswechsel**. Dieser kann zu jedem beliebigen Zeitpunkt stattfinden. Die kleine Routine des Betriebssystems, die für das Umschalten des Prozessors zwischen den Prozessen zuständig ist, nennt man **Dispatcher**. Ihre Aufgabe ist:

Prozesswechsel

Dispatcher

- Sichern der Informationen über den bisher rechnenden Prozess in dessen Prozesskontrollblock.
- Übergeben dieses Prozesskontrollblocks an den Scheduler (s. u.) zum Einfügen in die entsprechende Warteschlange (*bereit*, *blockiert*) ein.
- Wiederherstellen des alten Zustands des von Scheduler ausgewählten nächsten Prozesses aus dessen Prozesskontrollblock.
- Den Prozessor an den neuen Prozess übergeben, damit dieser in den Zustand *rechnend* kommt.

Bei heute üblichen Systemen schaltet das Betriebssystem den Prozessor viele Male pro Sekunde zwischen verschiedenen Prozessen hin und her, siehe Abbildung 2.4. Es ist klar, dass der Dispatcher deshalb so schnell wie möglich arbeiten muss. Da der Dispatcher auch direkt auf die Prozessorregister zugreifen muss, ist er in der Regel in der Maschinensprache (Assembler) programmiert.

Scheduler

Welcher Prozess als nächster den Prozessor erhalten soll, wird vom **Scheduler** festgelegt, siehe dazu Abschnitt 2.5.

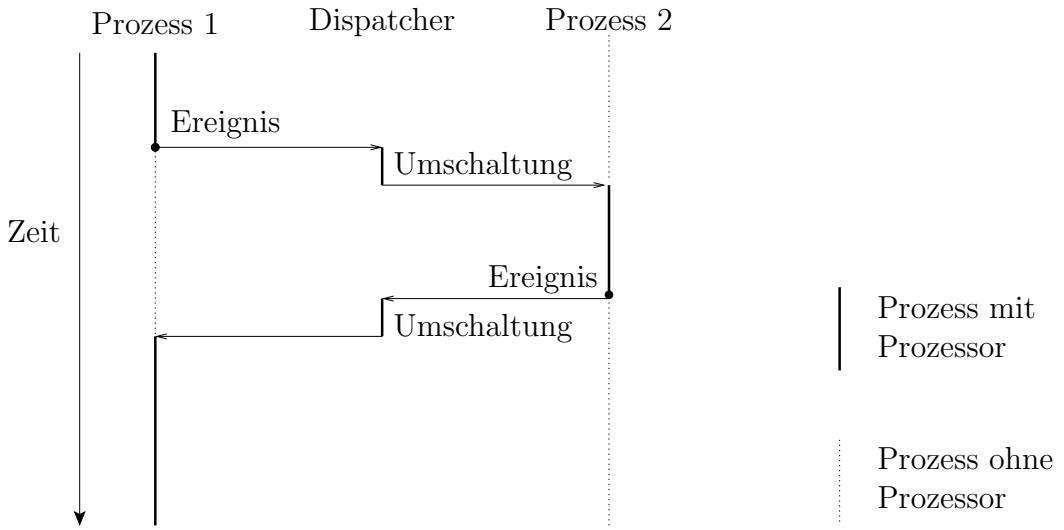

Abbildung 2.4: Ablauf der Prozessorumschaltung.

In Systemen, die nicht-präemptiv arbeiten, wird der Dispatcher vom vorher rechnenden Prozess aufgerufen. Bei präemptiven Systemen ist zunächst nicht klar, wie einem rechnenden Prozess der Prozessor gegen dessen Willen entzogen werden kann. Die Lösung dieses Problems bietet die Hardware. Moderne Rechner haben einen **Timer** (Zeitschalter), der zu Beginn jedes rechnenden Prozesses gesetzt wird und jeweils nach Ablauf einer bestimmten Zeitscheibe (z. B. 100 ms) eine Unterbrechung (Interrupt) erzeugt. Jetzt hat das Betriebssystem den Prozessor wieder und kann den Dispatcher aufrufen und dadurch den Prozessor einem anderen Prozess zuteilen.

Timer

Wenn eine Prozessumschaltung stattfindet, muss eines der folgenden Ereignisse aufgetreten sein:

- Ein Prozess will einen **Systemaufruf** durchführen, z. B. eine Ein-/Ausgabe oder die Erzeugung eines neuen Prozesses. Der Prozess geht dann vom Zustand *rechnend* in den Zustand *blockiert* über. Systemaufruf
- Ein Prozess gibt den Prozessor ab, oder die Zeitscheibe ist abgelaufen und dadurch wird ein **Timer-Interrupt** ausgelöst, siehe dazu auch Abschnitt 2.5.3. Timer-Interrupt
- Ein Prozess ist mit der Abarbeitung seines Programms fertig und geht daher in den Zustand *beendet* über.
- Eine **Unterbrechung** wird durch ein Ein-/Ausgabegerät ausgelöst, d. h. ein Ereignis tritt ein, auf das ein Prozess im Zustand *blockiert* wartet. Der bisher blockierte Prozess wechselt in den Zustand *bereit*. Unterbrechung
- Der rechnende Prozess verursacht einen Fehler, z. B. Division durch Null, und ein **Trap** wird ausgelöst. Trap
- Ein **Seitenfehler** passiert, d. h. ein Prozess greift auf eine Speicherseite zu, die sich nicht im Hauptspeicher befindet, genaueres siehe Abschnitt 3.6.1. Seitenfehler

Die Aktionen bei einem Prozesswechsel sind also zusammengefasst folgende:

- Sichern des Prozessorzustands mit allen Registern incl. Programmzähler.
- Aktualisieren des Prozesskontrollblocks des laufenden Prozesses, der Zustand wird auf *bereit*, *blockiert* oder *beendet* gesetzt, statistische Informationen fortschreiben.
- Einreihen des Prozesses in die entsprechende Warteschlange (*bereit*, *wartend auf Ereignis x*).
- Auswählen eines Prozesses zur Ausführung, dies macht der Scheduler, siehe Abschnitt 2.5.
- Aktualisieren des Prozesskontrollblocks des neuen Prozesses, der Zustand wird auf *rechnend* gesetzt.
- Anpassen der Hauptspeicherverwaltungsstrukturen, siehe dazu Kurseinheit 3.
- Wiederherstellen des alten Prozessorzustands des neuen Prozesses aus dessen Prozesskontrollblock.

2.5 Scheduling

In den bisherigen Abschnitten wurde dargestellt, wie das Betriebssystem den Prozessor zwischen zwei Prozessen umschalten kann. Die Frage, die sich als nächstes stellt, ist: Welchem Prozess soll das Betriebssystem denn als nächstes

den Prozessor zuteilen? Auf einem großen Rechner mit mehreren Benutzern hat das Betriebssystem die Qual der Wahl, welcher der vielen Prozesse aus dem Zustand *bereit* als nächster den Prozessor zugeteilt bekommen soll.

Scheduler

Der Teil des Betriebssystems, der diese Aufgabe erfüllt, wird **Scheduler** genannt. Man spricht jedoch nicht nur im Zusammenhang mit der Prozessorzuteilung von Scheduling. Dieser Begriff wird immer dann verwendet, wenn ein knappes Betriebsmittel (in unserem Fall der Prozessor) einer Reihe von Bewerbern (in unserem Fall den Prozessen) gegenüber steht, und eine Auswahl unter den Bewerbern getroffen werden muss.

Ein Rechner kann ein Programm, bzw. einen Prozess nur dann abarbeiten, wenn das Programm im Hauptspeicher des Rechners steht. Bisher sind wir stillschweigend davon ausgegangen, dass alle Prozesse bereits im Hauptspeicher stehen. In der Realität ist das natürlich nicht der Fall. Programme sind auf Sekundärspeichern, i. A. Festplatten gespeichert. Wenn ein Benutzer also einen neuen Prozess startet, der ein bestimmtes Programm ausführen soll, so muss zunächst das Programm vom Sekundärspeicher in den Hauptspeicher geladen werden. In Abhängigkeit von der gewählten Hauptspeicherverwaltung (siehe Kurseinheit 3) gibt es hier verschiedene Möglichkeiten. Wenn ein Programm immer komplett im Hauptspeicher stehen muss, stellt sich wiederum ein Scheduling-Problem.

Welcher Prozess soll als nächstes in den Hauptspeicher geladen werden? Ein

Long-Term Scheduler

Long-Term Scheduler trifft diese Auswahl. Zur Zeit der Großrechner (60er und 70er Jahre) war dies der Operateur an der Konsole, der die Bänder in den Rechner einlegte. Da man in moderneren Betriebssystemen darauf verzichtet, immer das komplette Programm im Hauptspeicher zu haben, stellt sich dieses Scheduling-Problem in dieser Form heute nicht mehr.

Im Unterschied dazu bezeichnet man den Scheduler, der aus der Menge der Prozesse im Zustand *bereit*[3] den nächsten rechnenden Prozess auswählt, als

Short-Term Scheduler

Short-Term Scheduler.

In diesem Abschnitt werden einige Strategien vorgestellt, nach denen diese Auswahl getroffen werden kann. Um die verschiedenen Strategien besser vergleichen und bewerten zu können, stellen wir zunächst einige Bewertungsmaßstäbe vor. Die Vor- und Nachteile der einzelnen Maßstäbe werden dann im Anschluss an die Strategien diskutiert.

2.5.1 Qualitätsmaßstäbe von Scheduling-Strategien

Prozessorauslastung. Früher waren Rechner sehr teuer und mit das teuerste Bauteil war der Prozessor. Man optimierte die Scheduler also daraufhin, den Prozessor möglichst gut auszulasten, um möglichst viel der teuren Rechenzeit zu nutzen. Heute ist dieses Kriterium (hohe **Prozessorauslastung**) nicht mehr ganz so wichtig. Prozessoren stehen in großer Zahl und preiswert zur Verfügung, so dass eine komplette Auslastung kaum vorkommt bzw. auch gar nicht wünschenswert ist.

Prozessorauslastung

[3] Diese Prozesse, bzw. die entscheidenden Teile des Prozesses befinden sich schon im Hauptspeicher.

Durchlaufzeit. Hat man wenige oder keine interaktiven Programme, sondern mehr *Batch Jobs* (siehe auch Stapelbetrieb, Seite 33) laufen, die z. B. Compiler, große Simulationen wie die Wettervorhersage oder auch längere Berechnungen aus dem Maschinenbau sein können, dann will man das fertige Ergebnis möglichst schnell bekommen. Eine minimale **Durchlaufzeit**, also das Zeitintervall zwischen Start und Ende des Prozesses, ist hier das anzustrebende Ziel.

Batch Jobs

Durchlaufzeit

Durchsatz. Aus kaufmännischer Sicht betrachtet, möchte man, dass der Rechner möglichst viel leistet, d. h. möglichst viel Arbeit in vorgegebener Zeit erledigt. Dieses Kriterium nennt man **Durchsatz**. Das Ziel des Schedulers ist also ein möglichst hoher Durchsatz, d. h. möglichst viele Aufträge sollen in einem bestimmten Zeitintervall bearbeitet werden. In diesem Fall ist der Begriff Auftrag mit einem Prozess gleichzusetzen.

Durchsatz

Antwortzeit. Der interaktive Benutzer eines Rechners aber wünscht sich möglichst sofortige Reaktionen auf eigene Eingaben, auch wenn viele Prozesse gleichzeitig vorhanden sind. Dagegen sind für ihn Kriterien wie kurze Durchlaufzeit oder hoher Durchsatz weniger interessant. Mit **Antwortzeit** bezeichnen wir das Zeitintervall zwischen Eingabe des Benutzers (z. B. Tastendruck oder Mausklick) und dem Beginn der Reaktion des Systems (z. B. Anzeige eines Zeichens oder kurzes Blinken). Eine kurze Antwortzeit ist also ein recht wichtiges Scheduling-Kriterium, weil heute immer mehr Aufgaben interaktiv am Rechner erledigt werden.

interaktiv

Antwortzeit

Fairness. Als einer unter vielen Benutzern will man natürlich auch, dass es bei der Verteilung der Prozessorkapazität gerecht zu geht. Niemand will u. U. ewig auf die Bearbeitung seines Auftrags warten, nur weil ständig jemand kommt und sich vordrängelt.

Fairness

Die folgende Liste zeigt noch einmal die bisher genannten Qualitätsmerkmale von Scheduling-Algorithmen:

Ziele beim Scheduling

- Maximale Effizienz, d. h. hohe Prozessorauslastung
- Minimale Antwortzeiten
- Minimale Durchlaufzeiten
- Maximaler Durchsatz
- Fairness, d. h. gerechte Verteilung des Prozessors

Alle genannten Kriterien (außer der Fairness) ändern ihre Werte im Laufe der Zeit. Die Prozessorauslastung ist manchmal recht niedrig, dann wiederum sehr hoch. Was soll also in diesem Zusammenhang der Begriff *maximale Effizienz* bedeuten?

In der Regel wird man versuchen, die *Durchschnittswerte* der Kriterien zu optimieren. Die durchschnittliche Antwortzeit soll minimal werden, die durchschnittliche Auslastung maximal. Bei zeitkritischen Systemen will man allerdings nicht eine minimale durchschnittliche Antwortzeit, sondern man will

höchstens einen bestimmten Zeitraum warten. Das *Maximum* der Antwortzeit soll also minimiert werden.

Es gibt Vorschläge, bei interaktiven Systemen die *Abweichung* von der durchschnittlichen Antwortzeit zu minimieren. Dadurch erhält man ein System, bei dem man die Antwortzeit gut vorhersagen kann. Für unerfahrene Benutzer ist dies u. U. eine große Hilfe, da sie sich sonst oft fragen: Was macht der Rechner so lange? Habe ich etwas falsch gemacht?

Optimale Scheduling-Algorithmen. Einen optimalen Scheduling-Algorithmus, der alle genannten Qualitätsmerkmale erfüllt, kann es aber nicht geben. Sorgt ein Scheduler für minimale Antwortzeiten, indem er alle Batch-Programme nur dann laufen lässt, wenn kein interaktiver Benutzer am Rechner ist, so verfehlt er das Ziel der minimalen Durchlaufzeit. Benutzer, die auf das Ende ihres Batch-jobs warten, werden benachteiligt und müssen vielleicht länger als nötig warten. Rechenzeit, die einer Benutzergruppe zugeschlagen wird, muss einer anderen Benutzergruppe weggenommen werden.

CPU burst. Bei der Betrachtung der Scheduling-Algorithmen ist nun nicht die komplette Rechenzeit eines Prozesses von Bedeutung. Während ihrer Abarbeitung geben viele Prozesse den Prozessor von sich aus ab, z. B. indem sie blockieren. Daher ist für das Scheduling nur die Zeit interessant, die ein Prozess den Prozessor am Stück behalten will. Diese Zeit wird **CPU burst** genannt. Interaktive Programme, die ständig Benutzereingaben erfordern, haben einen niedrigen CPU burst. Nach relativ kurzer Rechenzeit gehen sie in den Zustand *blockiert* über und geben dadurch den Prozessor wieder frei. Programme, die nur lange Berechungen ausführen, z. B. ein lineares Gleichungssystem mit 1000 Unbekannten zu lösen, haben dagegen einen großen CPU burst.

CPU burst

Wir gehen im Folgenden davon aus, dass die Prozesse ihre angegebene **Bedienzeit**, das ist die Zeit, in der sie den Prozessor brauchen, an einem Stück haben wollen, d. h. Bedienzeit = CPU burst. Mit **Wartezeit** bezeichnen wir die Zeit im Zustand *bereit*, da der Prozess auf den Prozessor wartet.

Bedienzeit
Wartezeit

In den folgenden Abschnitten wollen wir einige Scheduling-Verfahren genauer kennenlernen, die jeweiligen Vor- und Nachteile sind jeweils am Ende in einer Tabelle zusammengefasst.

2.5.2 Scheduling für nicht-präemptive Systeme

2.5.2.1 First Come First Served (FCFS)

FCFS

Eine einfache Scheduling-Strategie ist es, die Prozesse in der Reihenfolge ihrer Ankunft im System abzuarbeiten.

Zur Beschreibung dieses und auch der folgenden Verfahren soll immer wieder das folgende Beispiel herangezogen werden:

Gegeben sind die Prozesse P_1, P_2, P_3, P_4 und P_5. Sie treffen in dieser Reihenfolge zur selben Zeit im System ein. Die Bedienzeit der Prozesse ist in der folgenden Tabelle angegeben:

Prozess	P_1	P_2	P_3	P_4	P_5
Bedienzeit	22	2	3	5	8

Beim First Come First Served Scheduling werden die Prozesse in der Reihenfolge: P_1, P_2, P_3, P_4, P_5 abgearbeitet, siehe Abbildung 2.5.

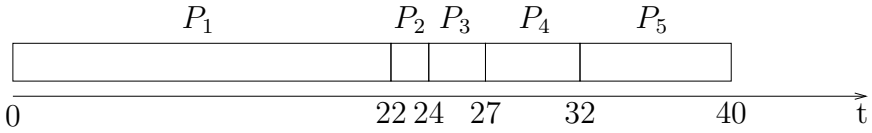

Abbildung 2.5: Ablauf bei *First Come First Served*.

Die Wartezeit R_i des Prozesses P_i für $i = 1, 2, 3, 4, 5$ ist die Zeit, die P_i in der Warteschlange verbracht hat. Also haben wir

$$\begin{aligned} R_1 &= 0 \\ R_2 &= 22 \\ R_3 &= 22 + 2 = 24 \\ R_4 &= 24 + 3 = 27 \\ R_5 &= 27 + 5 = 32 \end{aligned}$$

Die mittlere Wartezeit beträgt also:

$$R_\varnothing = \frac{22 + 24 + 27 + 32}{5} = \frac{105}{5} = 21$$

Ein kurzer Prozess hinter einem *Langläufer* muss nicht nur absolut, sondern auch relativ zu seiner eigenen Rechenzeit länger warten, als ein *Langläufer* hinter einem kurzen Prozess. Für interaktive Benutzer sind die dabei entstehenden Wartezeiten nicht zumutbar.

Dieser Scheduling-Algorithmus hat den Vorteil, dass er sehr leicht zu implementieren ist. Der Scheduler braucht alle bereiten Prozesse nur in einer normalen Warteschlange (queue) nach dem FIFO Prinzip zu verwalten. Der Zeitbedarf der Operationen Einfügen und Entfernen eines Elements in dieser Struktur ist konstant, also unabhängig von der Zahl der Prozesse, siehe auch die Zusammenfassung der Vor- und Nachteile der Strategie in Tabelle 2.1.

Vorteile	Nachteile
+ Einfach zu implementieren.	– kurz laufende Prozesse müssen u. U. sehr lange warten, daher für Dialogsysteme *nicht* geeignet.
+ geringer Verwaltungsaufwand, da die Operationen in konstanter Zeit arbeiten.	
+ Fairness, da alle Prozesse der Reihe nach den Prozessor erhalten.	

Tabelle 2.1: Eigenschaften von *First Come First Served*.

2.5.2.2 Shortest Job First (SJF)

SJF

Eine andere Scheduling-Strategie ist es, immer dem Prozess mit der kürzesten Bedienzeit den Prozessor zuzuordnen. Haben mehrere Prozesse dieselbe Bedienzeit, so kann man für diese Prozesse die First Come First Served Strategie anwenden.

In unserem oben schon genannten Beispiel ergibt sich folgende Ausführungsreihenfolge: P_2, P_3, P_4, P_5, P_1.

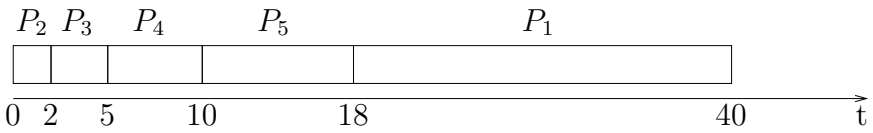

Abbildung 2.6: Ablauf bei *Shortest Job First*.

Die Wartezeiten der Prozesse sind:

$$\begin{aligned} R_1 &= 18 \\ R_2 &= \ 0 \\ R_3 &= \ 2 \\ R_4 &= \ 5 \\ R_5 &= 10 \end{aligned}$$

Daraus ergibt sich die mittlere Wartezeit zu:

$$R_{\varnothing} = \frac{2+5+10+18}{5} = \frac{35}{5} = 7$$

Minimalität

Man kann beweisen, dass diese Strategie immer die mittlere Wartezeit der einzelnen Prozesse minimiert.

Die Bedienzeit des ersten Prozesses (P_2) kommt bei allen folgenden Prozessen zu deren Wartezeit hinzu, d. h. in die durchschnittliche Wartezeit geht diese Bedienzeit mit dem Faktor $n-1$ (in unserem Fall 4) ein. Die zweite Bedienzeit (P_3) geht nur noch bei $n-2$ Prozessen, die dritte (P_4) nur noch

bei $n-3$ Prozessen, ..., die vorletzte Bedienzeit nur noch beim letzten Prozess in die Wartezeit ein. Die mittlere Wartezeit ist daher minimal, wenn die kleinsten Bedienzeiten bei den meisten Prozessen und umgekehrt, die größten Bedienzeiten bei den wenigsten Prozessen in die Wartezeit eingeht.

Ein großer Nachteil dieser Strategie wird offensichtlich, wenn regelmäßig ein neuer Prozess mit kurzer Bedienzeit hinzukommt. In unserem Beispiel könnte alle 3 Zeiteinheiten ein neuer Prozess mit der Bedienzeit 3 in das System eintreten. Ein bereits wartender Prozess mit größerer Bedienzeit wird dann *niemals* den Prozessor erhalten, da ja immer auch ein Prozess mit kürzerer Bedienzeit in der Warteschlange steht. Man spricht dann davon, dass der Prozess mit der längeren Bedienzeit **verhungert** (engl. **starvation**).

Verhungern
starvation

Um diese Strategie zu implementieren, kann man z. B. eine sortierte Liste für die bereiten Prozesse vorsehen. Das Entnehmen des nächsten Prozesses ist dann in konstanter Zeit möglich. Neu eintreffende Prozesse müssen dagegen einsortiert werden, was zu einer linearen Zeitkomplexität führt. An Stelle der linearen Liste kann man allerdings auch einen Heap benutzen. Das Einfügen eines Prozesses und auch das Entfernen sind dann in $O(\log n)$ Zeit möglich, wenn n die Anzahl der Prozesse ist. Die Eigenschaften der Strategie werden in Tabelle 2.2 zusammengefasst.

Implementierung

Das größte Problem bei der Implementierung dieser Scheduling-Strategie liegt darin, die Bedienzeit der Prozesse zu kennen. Früher musste man als Benutzer diesen Wert selbst schätzen und bei der Abgabe des Programms mitteilen. Beim Long-Term Scheduling ist diese Strategie also relativ leicht einsetzbar. Die Benutzer können bei immer wieder auftretenden Problemen, z. B. Lösung eines Gleichungssystems mit 200 Unbekannten, die Bedienzeit dieses Prozesses sehr gut vorhersagen. Bei unbekannten Programmen oder neuen Problemgrößen, bleibt aber auch dem erfahrenen Benutzer dann keine andere Möglichkeit, als zu raten.

Kenntnis der Bedienzeit?

Beim Short-Term Scheduling ist das Verfahren nicht so ohne weiteres anwendbar. Es gibt keine Möglichkeit, den Prozessen im Zustand *bereit* anzusehen, wie lang deren nächster CPU burst sein wird. Daher versucht man diesen Wert möglichst gut vorherzusagen. Anhaltspunkt für diese Vorhersage ist das bisherige Verhalten des Prozesses. Dabei geht man davon aus, dass ein Prozess, der bisher immer nur kurze CPU bursts hatte, auch in Zukunft mit großer Wahrscheinlichkeit nur kurze CPU bursts haben wird. Ein Prozess mit großen CPU bursts hat auch in Zukunft wahrscheinlich wieder große CPU bursts.

Vorhersage der Bedienzeit

Man benutzt dann die folgende Formel:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

Dabei ist τ_n die Schätzung des n-ten CPU bursts und t_n ist die tatsächliche Länge des n-ten CPU bursts. $0 \leq \alpha \leq 1$ ist ein Gewichtungsfaktor. Für $\alpha = 1$ ist nur der tatsächliche letzte CPU burst für die neue Schätzung maßgebend. Bei $\alpha = 0$ geht nur die letzte Schätzung in die neue Schätzung ein.

Übungsaufgabe 2.2 Gegeben sind die folgenden CPU bursts eines Prozesses: 4, 4, 4, 4, 6, 8, 10, 12, 12, 12, 12. Bestimmen Sie für $\alpha = 1/2$ und $\alpha = 3/4$ die Folgeschätzwerte, wenn die erste Schätzung 6 war.

Vorteile	Nachteile
+ Garantiert minimale Wartezeit für eine feste Menge von endlichen viele Prozesse mit bekannten Bedienzeiten.	– Langläufer müssen eventuell ewig warten.
+ Kurz laufende Prozesse werden *nicht* benachteiligt.	– Die Dauer des nächsten CPU bursts ist nicht bekannt und kann nur geraten bzw. geschätzt werden.
+ Algorithmus ist relativ leicht zu implementieren, falls die Dauer des nächsten CPU bursts bekannt ist.	– größerer Verwaltungsaufwand $O(\log n)$.
	– Prozesse können verhungern.

Tabelle 2.2: Eigenschaften von *Shortest Job First*.

2.5.2.3 Priority Scheduling

Die Idee des *Priority Scheduling* ist, dass man die Prozesse in unterschiedlich wichtige Klassen einteilt. Zuerst bekommen die Prozesse der wichtigsten Klasse (Prozesse mit der größten **Priorität**) den Prozessor zugeteilt, dann erst die der weniger wichtigen Klassen.

Priorität

Prioritäten können dabei *intern*, d. h. vom Betriebssystem selbst, oder *extern*, d. h. von außen, vorgegeben werden. Interne Prioritäten können z. B. vom Hauptspeicherbedarf, Plattenspeicherbedarf, der Anzahl der geöffneten Dateien, der bisher verbrauchten Rechenzeit usw. abhängen.

Externe Prioritäten sind häufig mit der Person des Benutzers verbunden. In einem Unternehmen hätten die Prozesse des Vorstands höchste Priorität, dann die Prozesse der Abteilungsleiter, Gruppenleiter, ... bis zuletzt die Prozesse der einfachen Mitarbeiter kommen. Eine andere Möglichkeit die Prioritäten zu vergeben liegt in der Bezahlung. Wenn die Benutzer für ihre Prozesse bezahlen müssen, so kann man die Preise nach Prioritäten staffeln. Wer es eilig hat, bekommt eine hohe Priorität für seine Prozesse und bezahlt dafür einen höheren Preis.

Die schon besprochene Shortest Job First Strategie kann man auch als eine Prioritäts-Strategie interpretieren. Je kleiner die Bedienzeit eines Prozesses ist, desto höher ist seine Priorität und umgekehrt.

In der Regel werden die Prioritäten durch natürliche Zahlen dargestellt, meist durch Zahlen aus einem festen Intervall. Einige Betriebssysteme verwenden kleine Zahlen für niedrige Priorität und große Zahlen für hohe Priorität. Es gibt aber auch Betriebssysteme, bei denen es gerade umgekehrt ist (kleine Zahl entspricht hoher Priorität). Im Kurstext sollen kleine Zahlen einer hohen Priorität entsprechen.

Wenn man die Prioritäten *statisch* festlegt, d. h. jeder Prozess bekommt eine Priorität zugeordnet und behält diese Priorität bis zu seinem Ende, hat man Probleme, den Prozessor gerecht zu verteilen, siehe auch Tabelle 2.3.

statische Priorität

Prozesse mit niedriger Priorität müssen u. U. ewig warten, falls ständig neue Prozesse mit hoher Priorität im System ankommen. Man kann dies verhindern, indem die Priorität der Prozesse *dynamisch* angepasst wird, siehe Feedback Scheduling.

dynamische Anpassung

Bei der Implementierung dieser Strategie braucht man wieder eine sortierte Liste, bzw. einen Heap. Das Einfügen und Entfernen eines Prozesses hat daher im Fall der sortierten Liste die Komplexität $O(n)$ und $O(1)$, bzw. beim Heap jeweils $O(\log n)$.

Übungsaufgabe 2.3 Erklären Sie, wie man einen Heap anstelle der sortierten Liste benutzt. Beschreiben Sie, wie der Prozess mit der größten Priorität entnommen wird, und wie ein neuer Prozess eingefügt wird.

Vorteile	Nachteile
+ *Wichtige* Aufgaben werden schnell erledigt.	– Bei statischer Prioritätsvergabe *nicht* fair, da Prozesse dann verhungern können.
	– Wie die Prioritäten festgelegt werden sollen, ist nicht so ohne weiteres klar.

Tabelle 2.3: Eigenschaften des *Priority Scheduling*.

Scheduling-Strategien für nicht-präemptive Systeme sind für den Dialogbetrieb nicht so gut geeignet. Ein bestimmter Prozess kann die Rechnernutzung für alle anderen unmöglich machen, indem er den Prozessor einfach nicht mehr abgibt.

Daher benutzt man im Dialogbetrieb lieber präemptive Scheduling-Verfahren, weil dabei den Prozessen der Prozessor entzogen werden kann.

2.5.3 Scheduling für präemptive Systeme

2.5.3.1 Round Robin

Eine für den interaktiven Betrieb (Dialogbetrieb) besser geeignete Scheduling-Strategie ist *Round Robin*. Die bereiten Prozesse werden, wie bei First Come First Served, in einer Warteschlange verwaltet. Der erste Prozess dieser Schlange bekommt als nächster den Prozessor zugeteilt.

Round Robin

Der rechnende Prozess darf den Prozessor jedoch nur für eine bestimmte Zeit behalten. Diese Zeit nennt man **Zeitscheibe** (engl. **time slice**) oder auch **Quantum**. Hat der rechnende Prozess den Prozessor nach dem Ablauf seiner Zeitscheibe noch nicht freigegeben, so wird er vom Betriebssystem unterbrochen und kommt dann an das *Ende* der Warteschlange. Dort werden auch neu entstandene Prozesse eingereiht.

Zeitscheibe
Quantum

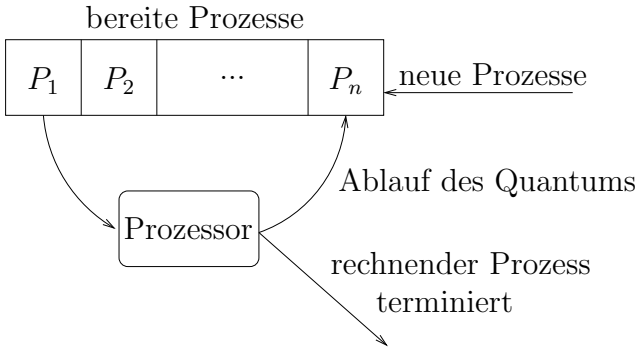

Abbildung 2.7: Schema bei *Round Robin*.

Die Größe der Zeitscheibe ist ein sorgfältig zu wählender Parameter dieser Strategie. Jeder Prozesswechsel dauert ja auch eine gewisse Zeit. Wählt man das Quantum q zu klein, z. B. nur so groß wie die Prozesswechselzeit, passiert folgendes: Ein Prozess rechnet q Zeiteinheiten, und dann findet der Prozesswechsel statt, der wiederum q Zeiteinheiten benötigt. Es sind $2q$ Zeiteinheiten vergangen, in denen der Prozessor aber nur q Zeiteinheiten für die Benutzerprozesse zur Verfügung stand. Die andere Hälfte der Zeit war der Prozessor mit der Verwaltungsarbeit des Betriebssystems (Prozessumschaltung) beschäftigt. Die Prozessorauslastung ist, was die eigentlich zu leistende Arbeit betrifft, zu gering.

Wählt man das Quantum allerdings zu groß, ändert sich zwar das Verhältnis echter Arbeit zu Verwaltungsaufwand zugunsten der echten Arbeit, aber die Wartezeiten für die interaktiven Benutzer werden unzumutbar groß. Falls ein Prozess gerade am Ende der Warteschlange steht und alle Prozesse vorher das größere Quantum komplett ausnutzen, können schnell Wartezeiten im Bereich von mehreren Sekunden auftreten. Man muss also für das Quantum einen geeigneten (mittleren) Wert finden. Eine sinnvolle Größe für das Quantum ist ein Wert, der etwas größer als die für eine übliche Interaktion erforderliche Zeit ist, typischerweise in Millisekunden-Größenordnungen.

Abbildung 2.8 zeigt den zeitlichen Ablauf unserer 5 Beispielprozesse, wenn das Scheduling nach der Round Robin Strategie vorgenommen wird. Das gewählte Zeitquantum beträgt 3 Zeiteinheiten. Die Prozessumschaltzeit wird hier mit 0 Zeiteinheiten angenommen.

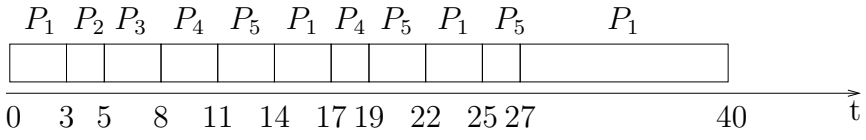

Abbildung 2.8: Der zeitliche Ablauf bei *Round Robin*.

Wenn ein Prozess sein Quantum nicht bis zum Ende ausnutzt, findet natürlich sofort ein Prozesswechsel statt. Wenn dagegen nur noch ein rechenbereiter Prozess vorhanden ist, braucht dieser nicht nach jedem Ablauf des Quantums unterbrochen zu werden, nur damit er den Prozessor sofort wieder erhält. Die einzelnen Wartezeiten und die Ausführungsreihenfolge der Prozesse

werden in Tabelle 2.4 dargestellt.

Prozess	Ausführungsreihenfolge										Wartezeit
	P_1	P_2	P_3	P_4	P_5	P_1	P_4	P_5	P_1	P_5	
P_1		2	3	3	3		2	3		2	18
P_2	3		-	-	-	-	-	-	-	-	3
P_3	3	2		-	-	-	-	-	-	-	5
P_4	3	2	3		3	3		-	-	-	14
P_5	3	2	3	3		3	2		3		19

Tabelle 2.4: Die Ausführungsreihenfolge der Prozesse und die Wartezeit der einzelnen Prozesse bei *Round Robin*.

Die Wartezeit der einzelnen Prozesse sind:

$$R_1 = 18 \quad R_2 = 3 \quad R_3 = 5 \quad R_4 = 14 \quad R_5 = 19$$

Die mittlere Wartezeit liegt also bei:

$$R_\oslash = \frac{3 + 5 + 14 + 18 + 19}{5} = \frac{69}{5} = 13{,}8$$

Verglichen mit der Strategie First Come First Served, hat sich die mittlere Wartezeit erheblich verbessert. Die Benachteiligung kurzer Prozesse findet bei diesem Verfahren nicht statt. Langlaufende Prozesse haben bei dieser Scheduling-Strategie sehr große Durchlaufzeiten, da kurzen Rechenzeiten (q) immer wieder lange Wartezeiten (Anzahl der Prozesse $\times$ q) gegenüber stehen.

Zur Implementierung des Verfahrens reicht wieder eine *normale* Warteschlange aus. Die Zeitkomplexität für das Einfügen und Entfernen eines Prozesses ist $O(1)$.

einfache Implementierung

Da alle Prozesse gleich behandelt werden (jeder bekommt sein Quantum des Prozessors), erfüllt diese Strategie das Qualitätsmerkmal der Fairness besonders gut, siehe auch die Zusammenfassung der Eigenschaften in Tabelle 2.5.

Fairness

Vorteile	**Nachteile**
+ Fairness, jeder Prozess bekommt seinen Anteil am Prozessor.	– Langlaufende Prozesse müssen bis zu ihrem Ende recht lange warten.
+ Kurz laufende Prozesse werden *nicht* benachteiligt.	
+ Einfach zu implementieren.	
+ Wenig Verwaltungsaufwand.	

Tabelle 2.5: Eigenschaften von *Round Robin*.

2.5.3.2 Shortest Remaining Time First

SRTF

Die vorhin besprochene Shortest Job First Strategie lässt sich einfach zu einer Strategie für präemptive Systeme erweitern, bei der auch rechnende Prozesse vom System unterbrochen werden können. Die Idee dabei ist, dass jedesmal, wenn ein neuer Prozess eintrifft, der rechnende Prozess angehalten wird. Die noch verbleibende Bedienzeit dieses Prozesses wird zu seiner neuen Bedienzeit. Dann wird wieder unter allen bereiten Prozessen, der Prozess mit der kleinsten Bedienzeit ausgewählt.

Anhand des folgenden Beispiels soll diese Strategie erklärt werden. Gegeben sind wiederum 5 Prozesse P_1, P_2, P_3, P_4 und P_5. Ihre Ankunftzeit im System und ihre Bedienzeit ist durch folgende Tabelle gegeben:

Prozess	P_1	P_2	P_3	P_4	P_5
Bedienzeit	22	2	3	5	8
Ankunftzeit	0	0	4	4	4

Zum Zeitpunkt 0 befinden sich nur die Prozesse P_1 und P_2 im System. Der kürzere der beiden Prozesse wird gestartet. Zum Zeitpunkt 2 ist P_2 beendet und nur noch P_1 im System. Also wird jetzt P_1 rechnend. Zum Zeitpunkt 4 wird der rechnende Prozess P_1 dann unterbrochen, da neue Prozesse im System eintreffen. P_1 hat dann noch eine verbleibende Bedienzeit von $22 - 2 = 20$ Zeiteinheiten. Da dies die größte Bedienzeit ist, erhalten zuerst die anderen Prozesse den Prozessor, danach dann wieder P_1 für seine restliche Laufzeit, siehe Abbildung 2.9. Die Wartezeit der einzelnen Prozesse sind:

$$\begin{aligned} R_2 &= 0 \\ R_3 &= 0 \\ R_4 &= 3 \\ R_5 &= 8 \\ R_1 &= 18 \end{aligned}$$

Die durchschnittliche Wartezeit beträgt also:

$$R_{\varnothing} = \frac{3+8+18}{5} = \frac{29}{5} = 5{,}8$$

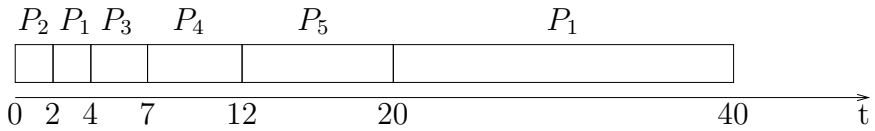

Abbildung 2.9: Ablauf bei *Shortest Remaining Time First*.

2.5.3.3 Priority Scheduling

Auch beim Priority Scheduling kann man sich eine Version für präemptive Systeme analog zur Shortest Job First Strategie überlegen. Immer wenn ein

neuer Prozess im System eintrifft, wird der rechnende Prozess unterbrochen, und danach erhält der Prozess mit der höchsten Priorität den Prozessor. Das kann dann entweder der bisher rechnende Prozess sein, falls der neue Prozess eine kleinere Priorität hatte, oder es ist der neue Prozess, der dann eine höhere Priorität als der bisher rechnende Prozess hat.

Übungsaufgabe 2.4 Gegeben sind die Prozesse $P_1, \ldots, P_5$. Sie treffen in dieser Reihenfolge zur selben Zeit im System ein. Die Prozesse haben folgende Bearbeitungszeiten: 15, 7, 1, 4, und 8. Geben sie für jeden der folgenden Scheduling Algorithmen die mittlere Wartezeit R an.

- First Come First Served
- Shortest Job First
- Round Robin mit $Q = 4$.

2.5.4 Kombinationen der Scheduling-Strategien

Die bisher betrachteten Scheduling-Strategien hatten jeweils ihre Vor- und Nachteile. Durch gleichzeitige Anwendung verschiedener Strategien versucht man nun, die unterschiedlichen Vorteile verschiedener Strategien zu vereinigen und die Nachteile zu unterbinden.

2.5.4.1 Feedback Scheduling

Ein Nachteil des Round Robin Verfahrens ist die absolute Gleichbehandlung *aller* Prozesse. Dadurch müssen auch Prozesse, die ihr Quantum nicht komplett ausnutzen, sondern ständig wegen Ein-/Ausgaben blockieren, lange auf die nächste Prozessorzuteilung warten. Bei solchen Prozessen ist es aber besser, ihnen den Prozessor so schnell wie möglich wieder zu geben. Dann kann der Prozess schnell seine nächste Ein-/Ausgabe starten, er blockiert wieder, und die Ein-/Ausgabe kann parallel zur Abarbeitung der anderen Prozesse stattfinden.

Man fasst Verfahren, die die Vergangenheit eines Prozesses bei der Auswahl des nächsten Prozesses berücksichtigen, unter dem Begriff **Feedback Scheduling** zusammen.

Feedback Scheduling

Beim Priority Scheduling kann man die Priorität der einzelnen Prozesse z. B. bei jeder Prozessumschaltung neu berechnen. Prozesse, die nicht *rechnend* waren, erhalten eine höhere Priorität. Ein Prozess kann jetzt nicht mehr ewig warten müssen, da seine Priorität beim Warten ja ständig steigt. Irgendwann hat auch dieser Prozess dann die höchste Priorität und bekommt den Prozessor zugeteilt. In der Literatur ist dieses Verfahren der Prioritätsanpassung unter dem Stichwort **aging** bekannt.

aging

Nimmt man nun noch das Round Robin Verfahren hinzu, so kann man folgende Strategie (rechenzeitabhängiges Feedback Scheduling) realisieren:

Neue Prozesse erhalten zunächst eine *hohe* Priorität und eine *kleine* Zeitscheibe. Hat der Prozess sein Quantum komplett genutzt, so erhält er eine niedrigere Priorität, aber sein Quantum verdoppelt sich. So müssen sehr lang

laufende Prozesse nicht ständig ihren Zustand wechseln. Stattdessen kommen sie seltener an den Prozessor und dürfen ihn dafür dann etwas länger behalten. Kurz laufende Prozesse haben eine höhere Priorität und werden dadurch von den Langläufern nicht behindert.

Der Verwaltungsaufwand kann bei diesem Verfahren jedoch recht groß werden. Bei *jedem* Prozesswechsel *wird* die ganze Liste der bereiten Prozesse durchgesehen, damit die neuen Prioritäten bzw. Zeitscheiben vergeben werden können. Das hat zur Folge, dass für n Prozesse der Zeitbedarf des Prozesswechsels linear, d. h. $O(n)$ ist, siehe auch Vor- und Nachteile der Strategie in Tabelle 2.6.

Vorteile	Nachteile
+ Prozesse werden nicht ein für allemal kategorisiert, sondern je nach ihrem Verhalten beurteilt.	– Höherer Verwaltungsaufwand, da Prozesse ständig neu beurteilt werden müssen.
+ Größere Gerechtigkeit, da kein Verhungern von Prozessen.	

Tabelle 2.6: Eigenschaften des *Feedback Scheduling*.

Übungsaufgabe 2.5 Um keinen Prozess länger als 500 ms warten zu lassen, programmiert ein Systementwickler das Round Robin Verfahren mit dynamischer Zeitscheibengröße. Bei n bereiten Prozessen wird die Zeitscheibe Q auf 500 ms/n gesetzt. Was halten Sie von dieser Scheduling-Strategie?

2.5.4.2 Mehrere Warteschlangen

Klassifizierung

Ein häufig benutztes Verfahren besteht darin, die Prozesse wie beim Priority Scheduling in verschiedene Klassen einzuteilen. Jede der Klassen hat dann eine eigene Warteschlange, für die jeweils ein eigener Scheduler zuständig ist (Multiple Queues). Ein mögliche Einteilung könnte z. B. so aussehen:

1. Die Klasse der Systemprozesse. Sie hat die höchste Priorität 1. Innerhalb der Klasse werden die Prozesse nach dem First Come First Served Verfahren ausgewählt.

2. Die Klasse der Dialogprozesse. Sie hat mittlere Priorität 2. Innerhalb der Klasse werden die Prozesse nach dem Round Robin Verfahren ausgewählt.

3. Die Klasse der Hintergrundprozesse (Batch Jobs). Sie hat niedrigste Priorität (≥ 3). Innerhalb der Klasse kommt das rechenzeitabhängige Feedback Scheduling zur Anwendung.

Das „Gesamtscheduling“ kann nun so aussehen, dass Dialogprozesse erst dann *rechnend* werden können, wenn kein Prozess aus der Klasse der Systemprozesse mehr *bereit* ist. Und erst wenn weder System-, noch Dialogprozesse bereit sind, kann ein Hintergrundprozess den Prozessor erhalten.

Ein andere Möglichkeit des Gesamtscheduling ist es, die Prozessorzeit auf die verschiedenen Klassen zu verteilen. Man kann 60 % der Zeit für Systemprozesse, 30 % für die Dialogprozesse und 10 % für die Hintergrundprozesse reservieren. Die Scheduler der einzelnen Klassen können dann diese Zeit an „ihre“ Prozesse vergeben. Wenn in einer Klasse keine rechenbereiten Prozesse mehr sind, wird die Zeit dieser Klasse natürlich den anderen Klassen hinzugeschlagen und verfällt nicht einfach.

Fairness

Bei der gezeigten Klasseneinteilung (in System-, Dialog-, Hintergrundprozesse) wird jeder Prozess einmal in eine Klasse eingeordnet und bleibt dann für immer in dieser Klasse. Man kann auch hier Feedback Strategien anwenden, um Prozesse auch zwischen den Klassen wechseln zu lassen.

Ein Prozess, der bisher ständig Benutzereingaben verlangte, kann auf einmal eine lange Berechnung durchführen. Gehörte er also vorher zu den interaktiven Prozessen, so ist jetzt ein Hintergrundprozess aus ihm geworden. Dementsprechend sollte dieser Prozess beim Scheduling jetzt auch anders behandelt werden.

Insgesamt ergeben sich bei diesem Verfahren viele mögliche Scheduling-Strategien, je nachdem:

- wieviele Klassen man definiert,
- welche Scheduling-Strategie man für die einzelnen Klassen vorsieht,
- nach welcher Strategie man die Klasse des nächsten Prozesses bestimmt,
- ob und wie Prozesse ihre Klasse ändern können,
- in welche Klasse neue Prozesse zuerst kommen.

Die Vor- und Nachteile dieser Strategien sind noch einmal in Tabelle 2.7 zusammengefasst.

Vorteile	**Nachteile**
+ Sehr gute Differenzierung zwischen Prozessen.	– Hoher Verwaltungsaufwand für das Scheduling.
+ Aufgrund der vielen Parameter sehr vielseitig einstellbar.	– Aufgrund der vielen Parameter ist es sehr schwierig, eine gute Einstellung zu finden.

Tabelle 2.7: Eigenschaften des Schedulings mit mehreren Warteschlangen.

2.5.4.3 Scheduling in Linux

Nun werfen wir noch einen Blick auf eine konkrete Scheduler-Implementierung in Linux.

$O(1)$-Scheduler

Der sogenannte $O(1)$-*Scheduler* wurde für den Linux-Kern 2.6 entwickelt. Sein Name bezieht sich darauf, dass der Scheduler unabhängig von der Anzahl der Prozesse eine konstante Laufzeit pro Auswahl-Vorgang benötigt. Er kann also mit einer im Prinzip beliebigen Zahl von Prozessen gleich schnell umgehen.

Jeder Prozess im System besitzt eine Priorität zwischen 0 und 139, kleinere Zahlen bedeuten höhere Priorität. Die ersten 100 Prioritäten von 0 bis 99 sind für den Realzeitbetrieb reserviert. Prozesse mit höherer Priorität haben ein längeres Zeitquantum. Die Zeitquanten liegen zwischen 10 und 200 Millisekunden, siehe Abbildung 2.10.

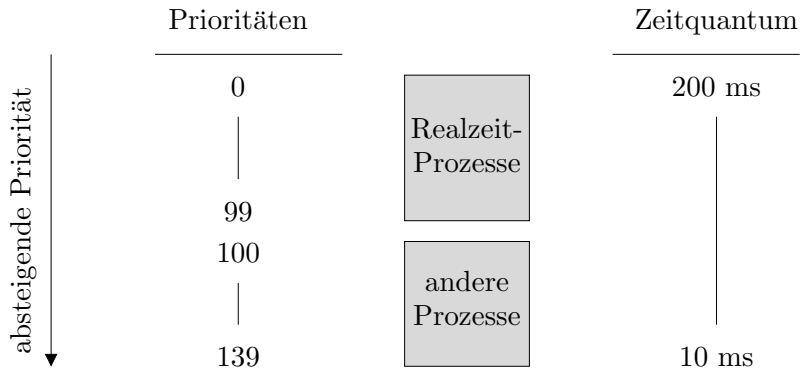

Abbildung 2.10: Prioritäten und Zeitquanten beim $O(1)$-Scheduler.

Die Realzeit-Prozesse bekommen statische Prioritäten, während die Prioritäten der anderen Prozesse abhängig von ihrem Verhalten immer wieder neu berechnet werden. Der $O(1)$-Scheduler bevorzugt interaktive Prozesse, indem die Priorität wegen häufiger Ein-/Ausgabeoperationen erhöht wird. Längeres Verweilen im Zustand *rechnend* erniedrigt dagegen die Priorität.

aktive Gruppe
abgelaufene Gruppe

Für die bereiten Prozesse gibt es für jede Priorität eine Warteschlange. Die Prozesse in einer solchen Warteschlange werden wieder in zwei Gruppen aufgeteilt, nämlich die aktive und die abgelaufene Gruppe. In der aktiven Gruppe befinden sich bereite Prozesse, die ihre Zeitquanten noch nicht aufgebraucht haben; die übrigen bereiten Prozesse sind in der abgelaufenen Gruppe.

Der $O(1)$-Scheduler wählt den Prozess in der aktiven Gruppe aus, der die höchste Priorität hat. Gibt es mehrere Prozesse mit der höchsten Priorität, dann folgt die Auswahl nach dem FIFO-Prinzip. Ist das Zeitquantum eines Prozesses aufgebraucht, berechnet der Scheduler seine Priorität neu, und er wird in die abgelaufene Gruppe verschoben. Der Prozess muss nun warten, bis alle Prozesse in der aktiven Gruppe ihre Zeitquanten aufgebraucht haben. Ist die aktive Gruppe leer, werden die beiden Gruppen vertauscht. Blockiert ein Prozess wegen einer Ein-/Ausgabe, geht er in die Warteschlange für blockierte Prozesse. Sobald das erwartete Ereignis eintritt, wird der Prozess aufgeweckt und geht wieder in die Warteschlange der aktiven Gruppe.

Übungsaufgabe 2.6 Kann ein Prozess mit der $O(1)$-Scheduling-Strategie verhungern, wenn wir annehmen, dass neue Prozesse in die abgelaufene Gruppe gehen?

2.5.5 Auswahl einer Scheduling-Strategie

Wie kann man nun aus der Menge der möglichen Scheduling-Strategien die optimale für ein konkretes System auswählen? In Abschnitt 2.5.1 wurden bereits einige Qualitätsmaßstäbe für die Scheduling-Strategien vorgestellt. Anhand dieser Maßstäbe kann man die verschiedenen Strategien vergleichen. Dazu gibt es verschiedene Vorgehensweisen.

Mathematischer Ansatz. Anhand von realen Daten misst man den CPU burst und die Ankunftzeiten von Prozessen. Aus diesen gemessenen Werten versucht man, die statistische Verteilung dieser Werte zu bestimmen. Man kann dann Formeln für die Wahrscheinlichkeit bestimmter Ankunftszeiten oder CPU bursts angeben. Aus diesen statistischen Verteilungen und Formeln kann man für die meisten Verfahren dann den durchschnittlichen Durchsatz, die Prozessorauslastung und die Wartezeit berechnen. So kann man die Strategie bestimmen, die bei dieser Verteilung z. B. die kleinsten Wartezeiten erreicht.

Oder man nimmt ein Beispiel von Prozessen mit typischen Werten der einzelnen Parameter (siehe das Beispiel beim First Come First Served Scheduling) und wendet die verschiedenen Strategien auf dieses Beispiel an. Dann kann man direkt die Zahlenwerte für die Qualitätsmerkmale ausrechnen.

Wenn man dieses Verfahren automatisieren will, dann schreibt man ein Programm, das eine Simulation der Strategien durchführt. Für größere und dadurch realistischere Beispiele kann man das Verhalten der verschiedenen Strategien durchspielen und sich die Qualitätsmerkmale ausrechnen.

Interpretationsprobleme. Alle bisher genannten Verfahren liefern entweder Zahlenwerte oder eine Formel. Wie diese Werte jedoch zu interpretieren und bewerten sind, bleibt ein Problem. Man kann zwar die Verfahren relativ zueinander bewerten, aber eine absolute Beurteilung ist nicht möglich. Wie will man z. B. beurteilen, ob eine durchschnittliche Antwortzeit von 500 ms tolerierbar ist oder nicht? Vielleicht ist die Antwortzeit ausgerechnet bei den Prozessen, die dem Benutzer wichtig sind, wesentlich schlechter. Außerdem ist es schwierig, sich von den Werten wirklich eine konkrete Vorstellung zu machen. (Empfindet man z. B. 500 ms als lang?)

Ein weiteres Problem ist, dass man nur von einem Modell der Wirklichkeit ausgeht. Man hat bestimmte Annahmen gemacht, die nicht exakt mit der Realität übereinstimmen müssen.

Realer Test. Diese Probleme umgeht man, indem man die verschiedenen Strategien in einem existierenden Betriebssystem implementiert und so unter den realistischsten Bedingungen (normaler täglicher Betrieb) testen kann.

Auch hier hat man aber das Problem, dass die Benutzer ihr Verhalten an das neue System anpassen werden. Während der Tests verhalten sich die Benutzer aber noch wie vorher gewohnt. Wenn man als Benutzer aber einmal

weiß, dass interaktive Prozesse bei der Prozessorvergabe stark bevorteilt werden, so wird man versuchen, auch die eigenen Batch Jobs wie interaktive Prozesse aussehen zu lassen. Man könnte z. B. einfach einige unsinnige Ausgaben machen, die man später einfach ignoriert. Das System könnte aber aufgrund der Ausgaben annehmen, dass es sich um einen interaktiven Prozess handelt und ihn bei der Vergabe des Prozessors bevorzugen.

Außerdem wird wahrscheinlich jeder Benutzer des Systems eine andere Rangfolge unter den Qualitätsmaßstäben haben. Dem einen sind optimale Durchlaufzeiten das Wichtigste, während andere Benutzer mehr Wert auf kurze Antwortzeiten legen. Allen Benutzern ist in der Regel zumindest der Wunsch nach schnellster Abarbeitung der eigenen Prozesse gemeinsam.

Zusammenfassend kann man also festhalten, dass es nicht einfach ist, für ein gegebenes System eine gute Scheduling-Strategie auszuwählen.

2.6 Vorteile und Probleme des allgemeinen Prozessmodells

Allgemeine Verwendbarkeit. Das bisher vorgestellte Modell der Prozesse ist sehr allgemein gehalten. Mit diesem Modell kann man parallele Aktivitäten auf einem Rechner sehr einfach beschreiben. Anhand eines Beispiels lässt sich das einfach nachvollziehen. Wenn ein Programmierer ein Programm erstellt, so braucht er dazu verschiedene Hilfsprogramme.

- Einen Editor, mit dem er den Programmtext schreibt.
- Einen Compiler, der das Programm in Maschinensprache übersetzt.
- Ein Programm, in dem er das Handbuch zum Betriebssystem ansehen kann.
- Einen Editor, mit dem er an der Dokumentation des Programms schreibt.
- Einen Debugger, unter dessen Kontrolle er den Ablauf seines Programms beobachten kann. So können Fehler im Programm einfacher gefunden werden.

Für jedes dieser Programme startet der Entwickler einen eigenen Prozess. Es ist z. B. sehr praktisch, wenn die Editor-Prozesse ständig laufen. Man kann jederzeit weiterschreiben, auch wenn gleichzeitig das Programm neu übersetzt wird. Zwischen den Tastendrücken im Editor wird der Prozessor dem Compiler-Prozess zugeteilt und kann dort weiter arbeiten. Wenn zwischendurch ein elektronischer Brief ankommt, so kann man einfach einen neuen Prozess starten, in dem der Brief gelesen wird.

Hoher Verwaltungsaufwand. Ein Prozess ist völlig getrennt von anderen Prozessen, er hat seinen eigenen Speicherbereich, eigene geöffnete Dateien, usw. Die Datenmenge, die das Betriebssystem im Prozesskontrollblock verwalten muss, ist recht groß.

Der Speicherbereich, der einem Prozess zugeordnet ist besteht in der Regel aus drei Segmenten, siehe dazu auch Abschnitt 2.2.2:

- Programmsegment, mit dem ausführbaren Programmcode.
- Stacksegment, mit dem Stapel der bei Prozeduraufrufen entsteht.
- Datensegment, mit den Daten, die vom Programm bearbeitet oder erzeugt werden.

Wenn der Entwickler zwei Editor-Prozesse laufen lässt, in denen er jeweils eine eigene Datei bearbeitet, so steht für jeden Prozess ein Editor-Programmsegment im Speicher. Da dieser Bereich immer gleich ist, kann man eine gemeinsame Benutzung dieses Speicherbereiches für verschiedene Editor-Prozesse vorsehen.

Es dauert trotzdem recht lang, bis ein neuer Prozess erzeugt ist.

Aufwändige Kommunikation. Da jeder Prozess komplett getrennt von allen anderen Prozessen abläuft, gibt es zunächst keine Möglichkeit der direkten Kommunikation zwischen den Prozessen. Indirekt, über den Inhalt von Dateien, können Informationen von einem Prozess zu einem anderen übertragen werden. Im obigen Beispiel sichert der Entwickler sein Programm, das er im Editor schreibt. Der Compiler-Prozess liest und übersetzt dann diese gesicherte Datei.

Für die Synchronisation ist der Anwender selbst verantwortlich. Wenn er während eines Compiler-Laufs die Datei im Editor ändert und abspeichert, kann er dadurch den Compiler-Prozess stören. Der Compiler könnte dann zur Hälfte die alte Version und zur Hälfte die neue Version der Datei lesen und übersetzen. Es ist klar, dass dabei die merkwürdigsten Fehlermeldungen des Compilers auftreten können. Weder der Compiler-Prozess, noch der Editor-Prozess können allein feststellen, dass dieser Fehler aufgetreten ist.

Daher bietet ein Betriebssystem weitere Möglichkeiten der Prozesskommunikation. In Kurseinheit 4 wird dieses Thema genauer behandelt.

Parallelität innerhalb von Prozessen. Ein Grund für die Erstellung des Prozessmodells war, dass man die Parallelität innerhalb eines Systems einfach beschreiben wollte. Parallelität tritt jetzt aber nur auf der Ebene der Prozesse selbst auf. Wenn man Parallelität innerhalb von Prozessen modellieren will, muss man weitere Anstrengungen unternehmen.

Die Probleme, die man mit dem bisherigen Modell der Prozesse hat, sind also:

– Recht großer Aufwand für Erzeugung neuer Prozesse.

– Prozesse belegen viele Betriebssystem-Ressourcen.

– Kommunikation zwischen den Prozessen nur indirekt oder über das Betriebssystem möglich.

– Parallelität innerhalb von Prozessen ist nur aufwändig zu beschreiben.

2.7 Leichtgewichtige Prozesse

Adressraum

Ausführungspfad

Jeder Prozess hat im bisherigen Prozessmodell einen eigenen Hauptspeicherbereich (*Adressraum*), auf den nur er selbst Zugriff hat. Der Adressraum enthält wie z. B. das Programm- und Datensegment, den Stack. Jeder Prozess hat einen einzigen **Ausführungspfad**, der durch die Inhalte von *Befehlszähler* und *Register* beschrieben wird.

Threads

Wenn wir uns nun vorstellen, dass mehrere Prozesse desselben Programmcodes zusammengelegt werden, so dass sie über einen *gemeinsamen Adressraum*, aber über jeweils einen eigenen Ausführungspfad verfügen, dann haben wir das Konzept der **Threads**[4]. Wenn ein Prozess also statt über einen über mehrere Ausführungspfade verfügt, dann besteht er aus mehreren Threads, siehe auch Abbildung 2.11.

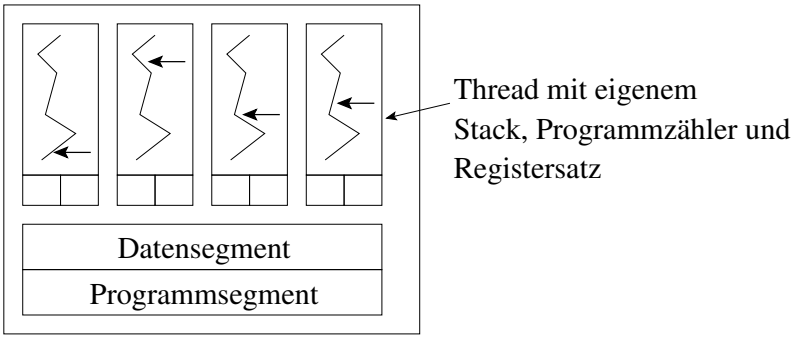

Abbildung 2.11: Ein Prozess mit mehreren Threads.

Trennung

Parallele Aktivitäten innerhalb eines Prozesses lassen sich dann dadurch einfach beschreiben. Ein Prozess kann aus mehreren Threads bestehen. Ein Thread ist durch seinen eigenen

- Registersatz
- Programmzähler
- Stackbereich

[4] thread (engl): Faden, Garn

definiert. Das *Programm- und Datensegment* teilen sich die Threads. Für die Synchronisation der Zugriffe auf die gemeinsamen Bereiche sind die Threads selbst zuständig. Abbildung 2.11 zeigt die Speichereinteilung bei einem Prozess, der aus mehreren Threads besteht. Das Betriebssystem hat hier keine Möglichkeit, den Zugriff eines Threads in z. B. den Stackbereich eines anderen Threads zu erkennen. Da diese Threads aber aus demselben Prozess stammen, und somit auch zu demselben Benutzer gehören, kann man davon ausgehen, dass solche Zugriffe kein Sicherheitsrisiko sind.

gemeinsamer Bereich
Synchronisation

Vorteile von Threads.

- *Effiziente Kommunikation*: Threads können direkt miteinander kommunizieren, indem sie das gemeinsame Datensegment für den Informationsaustausch verwendet.

- *Erreichbarkeit*: Wenn ein Prozess für eine bestimmte Aktion z. B. eine Anfrage nach einer Ressource blockiert würde, so muss nur ein Thread blockiert werden, während die anderen Threads weiterarbeiten können; siehe Abschnitt 2.7.1.

- *Teilen der gemeinsamen Ressourcen*: Es ist möglich, dass verschiedene Threads auf die gleichen Ressourcen, z. B. Datenspeicher, Prozessressourcen zugreifen.

- *Mehr Effizienz*: Threads haben gegenüber herkömmlichen Prozessen den Vorteil, dass sie effizienter arbeiten können, weil bei einem Wechsel der rechnenden Threads nur die Register ausgetauscht werden. Auch im Vergleich zum Erzeugen von weiteren echten Prozessen ist es günstiger, neue Threads zu kreieren.

2.7.1 Realisierungen von Threads

Man unterscheidet zwei Hauptarten von Thread-Realisierungen: **Benutzer-Threads** und **Kernel-Threads**.

Benutzer-Threads. Diese Realisierung ist die einfachere. Alle Thread-Informationen liegen im Benutzeradressraum, der Kern hat keinerlei Kenntnis davon, ob ein Prozess mehrere Threads verwendet oder nicht; siehe Abbildung 2.12. Um seine Threads zu verwalten, braucht jeder Prozess seinen eigenen privaten *Thread-Kontrollblock*, der analog zum Prozesskontrollblock ist. Er behandelt das Scheduling seiner Threads selbst mit Hilfe der entsprechenden Thread-Library-Funktionen, die auch für das Erzeugen und Beenden von Threads unabhängig vom Kern zur Verfügung stehen.

Benutzer-Threads

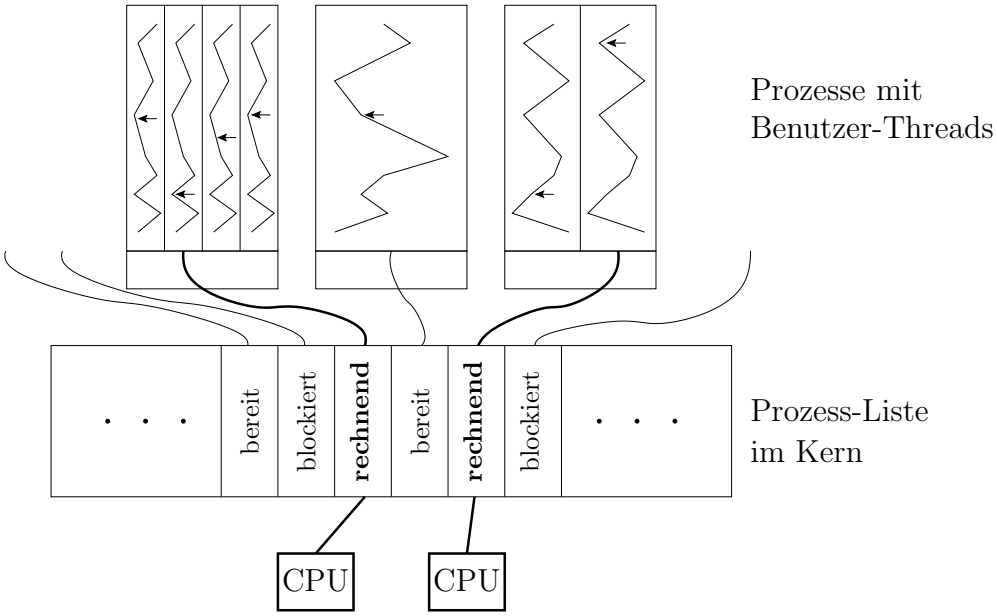

Abbildung 2.12: Zweiprozessor-System mit Benutzer-Threads.

Kernel-Threads

Kernel-Threads. Der Kern verwaltet die Threads ähnlich wie er schon Prozesse verwaltet. Es gibt eine Tabelle von Prozessen und Threads, die alle am Scheduling des Betriebssystem-Kerns teilnehmen; siehe Abbildung 2.13. Diese Realisierung ist erheblich aufwändiger, weil viele Systemaufrufe, die Prozesse betreffen, für Threads neu implementiert werden müssen.

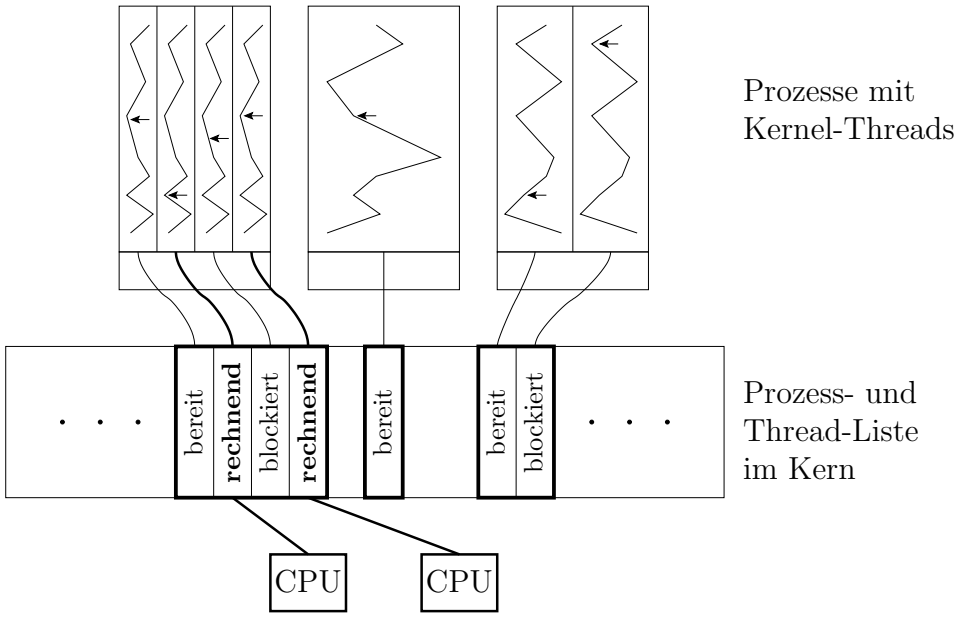

Abbildung 2.13: Zweiprozessor-System mit Kernel-Threads.

Vergleich der Thread-Realisierungen. Benutzer- und Kernel-Threads lassen sich am besten in einer direkten Gegenüberstellung vergleichen, die Vor- und Nachteile liegen mal auf der einen und mal auf der anderen Seite, siehe Tabelle 2.8.

	Benutzer-Threads		**Kernel-Threads**
+	Einfache Realisierung durch Bereitstellung einer Thread-Library auf Benutzerebene. Die Library lässt sich relativ leicht auf andere Betriebssysteme portieren.	−	Aufwändige Realisierung durch Neuimplementierung von Systemaufrufen.
−	Zusammengehörige Ben.-Threads können auch in Multiprozessor-Rechnern insgesamt nur einen Prozessor belegen und daher nur nacheinander abgearbeitet werden.	+	Zusammengehörige Kernel-Threads können in Multiprozessor-Rechnern auf mehreren Prozessoren parallel ablaufen.
−	Wenn ein Benutzer-Thread blockiert, dann ist der ganze Prozess mit allen zugehörigen Threads blockiert.	+	Ein blockierender Kernel-Thread eines Prozesses beeinflusst nicht die anderen Threads des Prozesses.
+	Das Erzeugen und Umschalten von zusammengehörigen Threads regelt der Prozess selbst. Erzeugung und Wechsel sind schneller, weil mit weniger Verwaltungsaufwand belastet, und flexibler, weil der Prozess seine eigene Zuteilungsstrategie realisieren kann.	−	Bei Erzeugung und Umschaltung von Kernel-Threads ist der Kern beteiligt und macht in etwa dasselbe wie bei normaler Prozesserzeugung bzw. normalem Prozesswechsel, daraus resultiert ein gewisser Verwaltungsaufwand.
−	Ein Benutzer-Thread kann nicht ohne besonders trickige Programmierung von seinem eigenen Prozess zu einer Unterbrechung gezwungen werden, was also ungefähr einem nichtpräemptiven System entspricht.	+	Die ausgeklügelte Scheduling-Strategie des Betriebssystems wird mitbenutzt, dadurch wird eine faire Behandlung aller Threads garantiert.
+	Ein Prozess mit sehr vielen Benutzer-	−	Ein Prozess mit sehr vielen Kernel-
−	Threads wird das Gesamtsystem kaum stärker belasten als einer mit wenigen. Andererseits wird ein solcher Prozess möglicherweise weniger CPU-Zeit bekommen als er sinnvollerweise bekommen müsste.	+	Threads hat starke Auswirkungen auf die Leistung des Gesamtsystems. Andererseits kann es natürlich auch erwünscht sein, dass das Gesamtsystem seine Leistung hauptsächlich auf diesen einen Prozess und dessen Threads konzentriert.

Tabelle 2.8: Eigenschaften der verschiedenen Thread-Realisierungen.

Realisierungen von Threads in aktuellen Betriebssystemen sind oft keine reinen Kernel-Threads, sondern Mischformen. Sie unterscheiden sich auch in der Art, wie Unterbrechungen bei Programmfehlern oder wie geöffnete Dateien behandelt werden oder was ein `fork`-Systemaufruf in einem Prozess mit mehreren Threads genau bewirkt. Dasselbe Programm wird sich deshalb auf verschiedenen Systemen durchaus unterschiedlich verhalten.

2.7.2 Anwendungsgebiete für Threads

Mehrprozessor-Maschinen. Wenn in einem Rechner mehr als ein Prozessor vorhanden ist, so kann der Programmierer mit Hilfe von Threads sein Programm „parallelisieren“ und dadurch wesentlich beschleunigen. Verschiedene Threads können jeweils einem eigenen Prozessor zugewiesen werden und dann parallel abgearbeitet werden.

Gerätetreiber für langsame Geräte. Ein Gerätetreiber für z. B. ein Diskettenlaufwerk ist ein Prozess, der immer eine Reihe von Anfragen zu erfüllen hat. Verschiedene Prozesse des Rechners wollen jeweils bestimmte Daten lesen. Jede dieser Anforderungen zerfällt im Prinzip in drei Teile.

1. Zunächst muss der Treiber den Auftrag in eine Liste eintragen, sowie die übergebenen Parameter analysieren und interpretieren. Hier findet dann auch die Berechung der Sektoradresse statt.

2. In der zweiten Phase findet die eigentliche Datenübertragung statt. Während dieser Phase ist der Treiberprozess blockiert, da er auf das Ende der Übertragung wartet. Es wäre an dieser Stelle aber schon möglich, mit der Bearbeitung des nächsten Auftrags zu beginnen und dort schon einmal die Parameter zu analysieren.

3. Am Ende kontrolliert der Treiber, ob alle Daten richtig übertragen wurden (Checksummen) und meldet dem Rechner, dass der Auftrag erledigt wurde.

Für jede Anfrage sind also dieselben Schritte durchzuführen. Es bietet sich an, für jede Anfrage einen eigenen Thread zu erzeugen. Der Prozessor im Controller kann dann schon die nächste Anfrage (Thread) bearbeiten, falls der Lese-/Schreibkopf für die vorherige Anfrage positioniert wird (d. h. der vorherige Thread ist im Zustand blockiert).

Verteilte Systeme. Auch hier hat man oft bestimmte Server (Dateiserver, Druckserver), die ihre Dienste den verschiedensten Clients (Anwendern) anbieten. Beispielsweise kann ein Client an einen Dateiserver eine Anfrage der Art „Gib mir die ersten 1000 Bytes der Datei mit dem Namen xyz.“ stellen. Es ist klar, dass auch hier verschiedene Anfragen vom Server soweit wie möglich parallel bearbeitet werden sollen. Threads sind eine einfache Möglichkeit, dies zu realisieren.

2.8 Zusammenfassung

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie:

- den Unterschied zwischen Programm und Prozess erklären können.
- die Prozesszustände und die möglichen Zustandsübergänge mitsamt ihren Ursachen verstanden haben.
- den Zweck und Inhalt eines PCBs beschreiben können.
- die Aufgaben des Dispatchers und des Schedulers angeben können.
- die vorgestellten Scheduling Strategien verstanden haben und die Arbeitsweise an Beispielen demonstrieren können.
- den Unterschied zwischen Threads und Prozessen kennen und jeweilige Anwendungsgebiete dieser Modelle angeben können.

Literatur

[NiBuPr96] B. Nichols, D. Buttlar, J. Proulx Farrell. *Pthreads Programming.* O'Reilly 1996.

[SiGaGa08] A. Silberschatz, P. B. Galvin, G. Gagne. *Operating System Concepts: Eight Edition.* John Wiley & Sons, Inc. 2008.

[Ta02] A. S. Tanenbaum. *Moderne Betriebssysteme: 2., überarbeitete Auflage.* Pearson Studium, 2002.

[Ta15] A. S. Tanenbaum and H. Bos. *Modern Operating Systems: Fourth Edition.* Pearson Education, 2015.

Lösungen der Übungsaufgaben

Übungsaufgabe 2.1, Seite 54: Wenn das Ereignis eintritt, auf das ein Prozess im Zustand blockiert wartet, ist in der Regel ein anderer Prozess im Zustand rechnend. Der bisher blockierte Prozess kann also den Prozessor nicht sofort erhalten (in den Zustand rechnend wechseln), da ein anderer Prozess den Prozessor hat. Der blockierte Prozess wechselt daher in den Zustand bereit und kann bei der nächsten Prozessorvergabe mit berücksichtigt werden.

Übungsaufgabe 2.2, Seite 63: Zur Berechnung der Werte wird die Formel

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

benutzt. Aus der ersten Schätzung von 6 und dem ersten CPU burst von 4 ergibt sich die nächste Schätzung zu:

$$\tau_{n+1} = \alpha \cdot 4 + (1 - \alpha) \cdot 6$$

Die folgende Tabelle zeigt die geschätzten Werte. Die Zeiten wurden dabei auf zwei Stellen nach dem Komma gerundet.

CPU burst	4.00	4.00	4.00	4.00	6.00	8.00	10.00	12.00	12.00	12.00	12.00
$\alpha = 1/2$	6.00	5.00	4.50	4.25	4.12	5.06	6.53	8.27	10.13	11.07	11.53
$\alpha = 3/4$	6.00	4.50	4.13	4.03	4.01	5.50	7.38	9.34	11.34	11.83	11.96

Übungsaufgabe 2.3, Seite 65: Ein Heap ist ein fast vollständiger binärer Baum, bei dem für jeden inneren Knoten x_i gilt: $\text{Key}(x_i) \leq \text{Key}(x_{2i}) \wedge \text{Key}(x_i) \leq \text{Key}(x_{2i+1})$. Dabei sind x_{2i} und x_{2i+1} die Kinder des Knotens x_i. Heaps lassen sich sehr leicht in arrays speichern.

In unserem Fall ist der Schlüssel gleich der Prioritätszahl. Der Knoten mit der kleinsten Zahl, d. h. mit der höchsten Priorität steht in der Wurzel des Baums. Wenn dieser Knoten entfernt wird, so wird das letzte Element des Heaps zur Wurzel gemacht. Dann vertauscht man dieses Element solange mit dem kleineren seiner Kinder, bis es unten angekommen ist oder kein kleineres Kind mehr existiert.

Im schlimmsten Fall muss man dabei die komplette Höhe des Baumes durchlaufen. Bei n Knoten ist die Höhe $\log n$.

Fügt man ein neues Element in einen Heap ein, so stellt man es zuerst an das Ende. Dann vertauscht man dieses Element so lange mit seinem Vater, bis es größer als der Vater oder an der Wurzel angekommen ist. Auch hier muss man evtl. den Baum in voller Höhe durchlaufen.

Genaueres hierzu können Sie im Kurs 1663 (Datenstrukturen) unter dem Stichwort priority queues nachlesen.

Übungsaufgabe 2.4, Seite 69: Die Wartezeit der Prozesse ist gleich der Wartezeit auf den Prozessor.

- Sei R_i die Wartezeit des Prozesses P_i für $i = 1, \ldots, 5$.

$$\begin{aligned} R_1 &= 0 \\ R_2 &= 15 \\ R_3 &= 15 + 7 = 22 \\ R_4 &= 22 + 1 = 23 \\ R_5 &= 23 + 4 = 27 \end{aligned}$$

  Damit ergibt sich die mittlere Wartezeit

$$R = \frac{15 + 22 + 23 + 27}{5} = \frac{87}{5} = 17,4.$$

- Die Prozesse werden in der Reihenfolge P_3, P_4, P_2, P_5, P_1 abgearbeitet. Daraus ergeben sich folgende Wartezeiten:

$$\begin{aligned} R_1 &= 12 + 8 = 20 \\ R_2 &= 1 + 4 = 5 \\ R_3 &= 0 \\ R_4 &= 1 \\ R_5 &= 5 + 7 = 12 \end{aligned}$$

  Die mittlere Wartezeit ist also

$$R = \frac{1 + 5 + 12 + 20}{5} = \frac{38}{5} = 7,6.$$

- Die folgende Tabelle zeigt die Ausführungsreihenfolge und Wartezeiten der einzelnen Prozesse beim Scheduling-Verfahren Round Robin mit Zeitquantum $Q = 4$.

Prozess	Ausführungsreihenfolge								Wartezeit
	P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_5	
P_1		4	1	4	4		3	4	20
P_2	4		1	4	4	4		-	17
P_3	4	4		-	-	-	-	-	8
P_4	4	4	1		-	-	-	-	9
P_5	4	4	1	4		4	3		20

  Die Summe der Wartezeit ist $20 + 17 + 8 + 9 + 20 = 74$. Daraus ergibt sich die mittlere Wartezeit zu $74/5 = 14.8$.

Übungsaufgabe 2.5, Seite 70: Solange das System nicht überlastet ist, arbeitet diese Strategie wie geplant.

Wenn allerdings zu viele Prozesse im Zustand bereit sind, so wird das Quantum immer kleiner. Deshalb werden immer mehr Prozessumschaltungen erforderlich. Eine Prozessumschaltung braucht eine konstante, in der Regel kleine Zeit. Wenn das Quantum aber wesentlich kleiner als die Prozessumschaltzeit wird, ist der Rechner nur noch mit Prozessumschaltungen beschäftigt und kommt kaum noch dazu, die Benutzerprozesse zu bearbeiten.

Wenn neben dem rechnenden Prozess kein weiterer Prozess mehr bereit ist, so ist das Zeitquamtum dann als 500ms/0 definiert. Diesen Fall (Division durch 0) muss der Scheduling Algorithmus abfangen und kann dann die Zeitscheibe auf einen vernünftigen großen Wert setzen.

Übungsaufgabe 2.6, Seite 72: Ein neuer Prozess gehört immer zu der abgelaufenen Gruppe und kann sich nicht in die aktive Gruppe vordrängen. Jeder Prozess in einer Warteschlange der aktiven Gruppe hat eine feste Priorität, die nur verändert werden kann, wenn der Prozess seine Zeitscheibe aufgebraucht hat. Danach wechselt er in eine Warteschlange der abgelaufenen Gruppe. Also kann ein Prozess in der aktiven Gruppe nicht von Prozessen in der gleichen Gruppe mit niedrigeren Prioritäten überholt werden. Da jeder Prozess ein endliches Zeitquantum hat, wird jeder Prozess in der aktiven Gruppe irgendwann in die abgelaufenen Gruppe verschoben, so dass die aktive Gruppe irgendwann keinen Prozess mehr hat. Weil dann die beiden Gruppen vertauscht werden, befindet sich jeder Prozess irgendwann in der aktiven Gruppe und wird auch garantiert irgendwann rechnend.

Glossar

Alterung (*aging*) heißt der Vorgang, bei dem die Priorität von Prozessen mit deren Alter(Wartezeit) steigt. Dadurch soll das Verhungern von Prozessen mit niedriger Priorität verhindert werden.

Antwortzeit (*response time*) Das Zeitintervall zwischen einer Eingabe und der ersten zu bemerkenden Reaktion in einem interaktiven System.

Bedienzeit (*service time*) Die zur Ausführung eines Auftrags nötige Prozessorzeit.

CPU burst Das Zeitintervall, für das ein Prozess den Prozessor ohne freiwillige Abgabe (auch durch I/O) behalten will.

Dispatcher (*dispatcher*) Der Teil des Betriebssystems, der die Umschaltung des Prozessors zwischen zwei Prozessen vornimmt.

Durchsatz (*throughput*) bezeichnet die „Leistung“ eines Systems. Sie ergibt sich als Quotient aus Arbeit, d. h. abgearbeiteten Prozessen, und dazu benötigter Zeit.

Feedback Scheduling ist eine Scheduling-Strategie, bei der das bisherige Verhalten eines Prozesses in das Scheduling eingeht.

First Come First Served ist eine Scheduling-Strategie, bei der die Prozesse in der Reihenfolge ihrer Ankunft den Prozesser zugeteilt bekommen.

long-term scheduler Der Teil des Betriebssystems, der entscheidet, welcher Prozess als nächster in das System, bzw. den Hauptspeicher geladen wird.

nicht-präemptiv heißt eine Scheduling-Strategie, die einem rechnenden Prozess den Prozessor nicht entzieht. Nur wenn der Prozess selbst den Prozessor abgibt (warten auf I/O oder freiwillige Abgabe des Prozessors), findet eine Prozessumschaltung statt.

präemptiv heißt eine Scheduling-Strategie, bei der das Betriebssystem einem rechnenden Prozess den Prozessor entziehen kann.

Priorität (*priority*) sie bestimmt die „Wichtigkeit“ von Prozessen und dadurch auch die Reihenfolge, in der sie den Prozessor zugeteilt bekommen.

Programm (*program*) Beschreibung eines Algorithmus in einer für Mensch oder Maschine verständlichen Sprache.

Prozess (*process*) Ein in der Ausführung befindliches Programm.

Prozessabbild (*process image*) Ein Prozessabbild besteht aus dem Programm, dem Stack, den Daten und dem Prozesskontrollblock eines Prozesses.

Prozesskontrollblock (*process control block*) Eine Datenstruktur, in der das Betriebssystem wichtige Daten über einen Prozess speichert.

Prozesstabelle (*process table*) Eine Datenstruktur für die Organisation von Prozessen im Betriebssystem.

Prozessorauslastung (*CPU utilization*) Der Anteil der Zeit, in der an Benutzerprozessen (d. h. keinen systeminternen Verwaltungsprozessen) gerechnet wird, an der Gesamtzeit.

Prozessumschaltung (*context switch*) Der Vorgang, bei dem der Prozessor von einem Prozess zu einem anderen Prozess umgeschaltet wird. Diese Aktion wird von Dispatcher ausgeführt.

Prozesszustand (*process state*) Die Beschreibung der aktuellen Aktivität eines Prozesses (Erzeugt, Bereit, Rechnend, Blockiert, Beendet).

Quantum Siehe Zeitscheibe.

Round Robin ist eine pre-emptive Scheduling-Strategie, bei der die Prozesse nacheinander den Prozessor für ein bestimmtes Quantum zugeteilt bekommen.

Scheduler Ein Teil des Betriebssystems, der aus einer Menge von Bewerbern um ein Betriebsmittel einen auswählt, der das Betriebsmittel dann zugeteilt bekommt. Am häufigsten im Zusammenhang mit der Prozessorzuteilung benutzt.

short-term scheduler Der Teil des Betriebssystems, der aus der Liste der bereiten Prozesse, den Prozess auswählt, der als nächstes den Prozessor zugeteilt bekommt.

Shortest Job First Scheduling-Strategie, bei der immer der Prozess mit der kürzesten Bedienzeit als nächster den Prozessor erhält.

Thread Leichtgewichtiger Prozess.

verhungern (*starvation*) Ein Prozess verhungert, wenn er zwar bereit ist, aber trotzdem niemals den Prozessor erhält und rechnen kann.

Wartezeit (*waiting time*) Die Zeit, für die ein Auftrag im System ist, aber nicht bearbeitet wird. Es ist die Zeit, die ein Prozess in der Bereit-Liste verbringt.

Zeitscheibe (*time slice*) Die Zeit, für die ein Prozess den Prozessor maximal belegen darf, bevor er vom System unterbrochen wird.