Planbarkeitsanalyse von Ada-*tasks* in Echtzeitsystemen

Diplomarbeit
von

Oliver Bühn

29. Juli 1999

Inhaltsverzeichnis

Bilderverzeichnis 5

1 Einleitung 7

2 Begriffsbestimmung 9
2.1 Echtzeit 9
2.2 Echtzeitbedingung 9
2.3 Echtzeitsystem 10
2.4 Tasks 11
2.4.1 Periodische *tasks* 14
2.4.2 Nichtperiodische *tasks* 14
2.5 Laufplanung 14
2.6 Planbarkeitsanalyse 16
2.6.1 Probleme bei der Planbarkeitsanalyse 18
2.7 Die Programmiersprache Ada 24

3 Planbarkeitsanalysemethoden 25
3.1 Prozessorbelastung 26
3.2 Kritische Zeitzone 27
3.3 Zugriff auf Betriebsmittel 28
3.4 Ratenmonotone Analyse (RMA) 33
3.5 Analyse für das ,,Planen nach Fristen" (EDFA) 36
3.6 Analyse der Antwortzeiten (RTA) 36

4 Eingabe für den Planbarkeitsanalysator 39

5 Ausgabe des Planbarkeitsanalysators 43

6 Entwurf des Planbarkeitsanalysators 46
6.1 Aufgabe des Planbarkeitsanalysators 46
6.2 Eingabe des Planbarkeitsanalysators 47
6.3 Ausgabe des Planbarkeitsanalysators 48
6.4 Programmiersprache 48

7 Umsetzung des Entwurfs 49
7.1 Wichtige Variablen 49
7.2 Einlesen der Eingabe 50
7.3 Analyse 53

8 Benutzung des Planbarkeitsanalysators 57
8.1 Testbeispiele 58

9 Ergebnisse dieser Arbeit **73**

10 Ausblick **73**

Literaturverzeichnis **77**

Stichwortverzeichnis **79**

Bildverzeichnis

1 Kenngrößen von *tasks* . . . 13
2 *pipelining* . . . 21
3 Prioritätenumkehrung . . . 23
4 kritische Zeitzone, kritischer Moment . . . 27
5 Prioritätenerben . . . 29
6 Tabelle für die Errechnung der Blockierzeiten . . . 30
7 maximale Antwortzeit . . . 37
8 Bedeutung der Angaben in der Spezifikationsdatei . . . 43
9 Analyseumgebung . . . 47
10 kritische Zeitzone von `rma1_test.dat` . . . 60
11 Laufplan der ersten 14 Zeiteinheiten für `rma_test3.dat` . . . 63
12 *RM* und *EDF* im Vergleich . . . 65

1 Einleitung

Echtzeitanwendungen finden heute mehr und mehr Einsatzgebiete. Waren solche Gebiete bisher klein und abgegrenzt, sind sie nun unüberschaubar. Durch fortschreitenden Preisverfall der elektronischen Bauelemente wird mit aller Macht versucht, diese auch überall zu integrieren.

Digitale Rechner im Echtzeitbetrieb sind zum Beispiel in Flugzeugsteuerungen, Fernsehapparaten, Kernkraftwerken, Verkehrsleitsystemen, Telekommunikationsanlagen, robotergesteuerten Fertigungsstrecken oder Luftraumüberwachungsanlagen zu finden. Überall dort, wo Rechner nur eine bestimmte Zeit für ihre Arbeit haben, meistens dort, wo ein schnelles Ergebnis erwartet wird, sind Echtzeitsysteme gefragt.

Im allgemeinen wird davon ausgegangen, daß man sich auf rechnergestützte Systeme verlassen kann, zumindest was ihre Zuverlässigkeit in Bezug auf die Richtigkeit der Ergebnisse angeht. Die Ergebnisse, die Echtzeitsysteme produzieren, sind nur richtig, wenn sie auch innerhalb einer Zeitvorgabe vorliegen. Das im voraus zu garantieren, ist eine schwierige Sache.

In den meisten Fällen in der heutigen Praxis werden Echtzeitsysteme eingesetzt, die zuvor nicht gründlich auf ihr zeitliches Verhalten analysiert worden sind. Vielmehr werden einfach die Rechenanlagen, auf denen diese Systeme laufen sollen, in Rechenleistung und Speichermöglichkeit überdimensioniert. Dann wird erwartet, daß die Systeme dadurch korrekt arbeiten werden. Meistens wird das auch gelingen.

Eine Garantie für zeitlich korrektes Arbeiten von Echtzeitsystemen können aber nur entsprechende Analyseverfahren geben, die auf Analysetheorien beruhen, die es schon seit vielen Jahren gibt. Leider sind diese Theorien in der Praxis kaum bekannt, so daß davon selten Gebrauch gemacht wird. In ausgewählten Einrichtungen, wie etwa in Kernkraftwerken, werden spezielle Analysemethoden für die Systemplanung eingesetzt. Einen universellen Analysator, wie er sicherlich wünschenswert wäre, gibt es noch nicht.

Weil sich langsam in der Industrie herumspricht, daß da Möglichkeiten existieren, mit denen die Zuverlässigkeit von Echtzeitsystemen bestimmt werden kann, werden gewisse finanzielle Mittel in die Forschungsgebiete gebracht, die Ergebnisse für einen Erfolg auf diesem Gebiet, der sich im späteren Absatz solcher Systeme bemerkbar machen kann, versprechen.

So hat zum Beispiel eine deutsche Telekommunikationsfirma Geld dafür ausgegeben, daß sie verwendbare Ergebnisse der zeitlichen Analyse von Ada-Anwendungen erhält. Damit soll der Firma möglich sein, ihre Anwendungen, die in der Programmiersprache Ada geschrieben sind, zu optimieren und von ihnen zu garantieren, daß sie zuverlässig in Echtzeitumgebungen laufen.

Dafür wurde ein Projekt unter dem Namen „ADAM“ begonnen. Der Name soll daran erinnern, daß Ada-Multitasking-Methoden zu untersuchen sind. Ziel dieses Projektes ist es, Ada-Anwendungen auf ihr späteres zeitliches Verhalten hin zu untersuchen. D. h. daß die Anwendungen analysiert werden, bevor sie zur Ausführung kommen. Damit ist es möglich, schon bei der Entwicklung abzusehen, an welcher Stelle es zu zeitbedingten Schwierigkeiten kommen könnte.

Gegenstand der vorliegenden Arbeit ist ein Teil einer Analyseumgebung, mit dem man im voraus bestimmen kann, ob sich eine Anwendung, genauer, die einzelnen Teile einer Anwendung, in vorher festgelegten zeitlichen Grenzen halten können oder nicht.

Zunächst werden grundsätzliche Begriffe geklärt, die für das weitere Verständnis der Arbeit wichtig sind. Anschließend wird eine Einführung in die Planbarkeitstheorie gegeben, bevor der Planbarkeitsanalysator, der praktische Gegenstand der Arbeit, erklärt wird.

2 Begriffsbestimmung

Im folgenden werden Begriffe erläutert, die in der weiteren Arbeit benutzt werden.

2.1 Echtzeit

Mit zunehmender Popularität der elektronischen Rechenanlagen wächst auch die Menge der Menschen, die sich des zugehörigen Jargons bedienen. Leider jedoch werden Begriffe oft verwechselt oder gänzlich mißverstanden. Ein gutes Beispiel ist der Begriff der **Echtzeit**.

Im allgemeinen wird angenommen, daß etwas in Echtzeit passiert, wenn es beim Beobachten zu keinen Störungen kommt. Ein Spiel also, daß nahezu kontinuierlichen Graphikaufbau bietet und bei dem ein Fahrzeug etwa sehr schnell manövriert werden kann, wird oft als Echtzeitspiel bezeichnet.

Tatsächlich werden bei den meisten Simulationsprogrammen zeitabhängige Berechnungen eine wichtige Rolle spielen. Zu verwendende Funktionen werden nur dann als korrekt angesehen, wenn sie nicht nur ein korrektes Ergebnis liefern, sondern dieses Ergebnis innerhalb einer Zeitvorgabe bereitstellen.

Diese Zeitvorgabe kommt häufig aus der natürlichen, echten Umgebung. Das System muß mit Zeitangaben aus der echten Umgebung umgehen können. Es muß daher ein Zeitmodell verwalten, was der echten Zeit, auch Echtzeit genannt, sehr nahe kommt.

Läuft die innere Zeit (Systemzeit) genauestmöglich parallel zum äußeren, echten Zeitfluß, so daß beliebige Zeitabschnitte, z. B. im Milli- oder gar Nanosekundenbereich, gemessen werden können, dann ist damit eine Voraussetzung gegeben, das System im Echtzeitbetrieb arbeiten lassen zu können.

2.2 Echtzeitbedingung

Die Echtzeitbedingung fordert, daß die Ergebnisse der Verarbeitung der von einem Ereignis anfallenden Daten innerhalb bestimmter Zeitvorgaben vorliegen müssen. Diese Bedingung wird in zwei Grade unterteilt: **harte** und **weiche Echtzeitbedingung**.

Bei der harten Variante wird erwartet, daß in jedem Fall das Ergebnis zum angegebenen Zeitpunkt vorliegt. Etwaige Verstöße können nicht akzeptiert werden.

Die weiche Echtzeitbedingung läßt zu, daß hin und wieder die Frist überschritten wird. Systeme mit solchen weichen Bedingungen verursachen keine Katastrophe, wenn das Ergebnis etwas zu spät präsent ist.

Ein Beispiel für eine Anwendung mit harter Echtzeitanforderung ist die Steuerung der Bremsanlage moderner Limousinen. Tritt der Fahrer auf das Bremspedal, welches mechanisch von der Bremse getrennt ist, sollte sofort eine Bremsreaktion erfolgen. Der verwendete Rechner und die darauf installierten Programme müssen in jedem Fall in der Lage sein, das zu ermöglichen.

Für eine Anwendung mit weicher Echtzeitanforderung sei die Bilddatenübertragung ein Beispiel. Werden bei einer Bildkonferenz bewegte Bilder der Gesprächspartner aus einem entfernt gelegenen Büro auf dem Empfängerbildschirm angezeigt, ist es höchstens etwas unangenehm, hat aber keinesfalls katastrophale Auswirkungen, wenn hin und wieder mal ein Bild gestört oder gar nicht angezeigt wird.

Der übertragene Ton allerdings sollte störfrei ankommen, da hier die Informationsdichte wesentlich höher ist als beim Bild und eine nur sehr kleine Auslassung eines gesprochenen Textes seinen Sinn stark verändern könnte. Hier wäre eine harte Echtzeitanforderung zu stellen.

Die Echtzeitbedingung besagt nicht, daß Ergebnisse immer besonders schnell vorliegen müssen. Eine weitere Anforderung an ein Echtzeitsystem ist die Vorhersagbarkeit des Zeitverhaltens. Es nützt nichts zu wissen, daß das System oder der Rechner, auf dem das System installiert ist, besonders schnell ist. Man muß bestimmen können, was zu einer beliebigen Zeit geschieht.

2.3 Echtzeitsystem

Ein **Echtzeitsystem** zeichnet sich dadurch aus, daß es seine Programme ständig betriebsbereit hält, d. h. daß diese Programme immer in der Lage, sind auf äußere Ereignisse zu reagieren, noch während diese auftreten.

Das Auftreten der Ereignisse kann zu beliebigen Zeitpunkten der echten Zeit der Umgebung geschehen. Die Ereignisse können sowohl periodisch als auch nichtperiodisch auftreten.

An ein Echtzeitsystem ist die Echtzeitbedingung zu stellen. Kann diese Bedingung nicht erfüllt werden, ist das System nicht in der Lage, im Echtzeitbetrieb zu arbeiten.

Viele **Echtzeitbetriebssysteme** basieren auf modifizierten Kernen von *time sharing*-Betriebssystemen. Dadurch haben diese dieselben Grundeigenschaften wie die ursprünglichen Systeme. Diese Eigenschaften genügen meist nicht den Anforderungen, die an Echtzeitbetriebssysteme zu stellen sind. Hauptmerkmale von Echtzeitbetriebssystemen sind:

- **Multitasking:** Ein Programm, das sich im Speicher befindet und bereit ist, ausgeführt zu werden oder ausgeführt wird, bezeichnet man auch als **Prozeß** (engl. *task*). Das nebenläufige Abarbeiten von Prozessen ist für ein Echtzeitsystem unerläßlich. Multitasking ist eine Voraussetzung dafür.
- **sofortiges Reagieren auf externe Ereignisse:** Wenn ein Ereignis aus der Umgebung auftritt, muß darauf in den meisten Fällen sofort reagiert werden. Um eine unmittelbare Reaktion zu ermöglichen, wird der Prozeß, der auf dieses Ereignis reagieren soll, bevorzugt ausgeführt. Das Echtzeitbetriebssystem muß in der Lage sein, einige Prozesse zu bevorzugen.
- **Mechanismen zur Prozeßkommunikation und -synchronisation:** Häufig werden Ergebnisse eines Prozesses als Ausgangswert für Berechnungen eines anderen Prozesses benötigt. Damit diese Werte von einem Prozeß zum anderen gereicht werden können, muß es Mechanismen für die Kommunikation zwischen den Prozessen geben. Diese schließen oft auch eine Synchronisation der Prozesse ein, damit die Werte zur richtigen Zeit übergeben werden können.
- **Darstellung von Echtzeit:** In Echtzeitbetriebssystemen werden Echtzeituhren eingesetzt, die den echten Zeitfluß der Umgebung möglichst genau registrieren. Dabei wird die interne Zeit häufig als eine Zahl geführt, die in möglichst kleinen Abständen erhöht wird.
- **gesonderte Benutzung der Ein-/Ausgabegeräte:** Die Benutzung der Ein- und Ausgabegeräte, wie zum Beispiel Festplatten oder Drucker, verhindert oft deterministisches Zeitverhalten. Es kommt dann zu Unterbrechungen des Betriebes, wenn auf derartige Geräte zugegriffen wird. Es sollte möglich sein, entweder eigene Gerätetreiber, die zeitkritische Abläufe anderer Prozesse nicht gefährden, zu implementieren, oder die vorhandenen zu verändern. In manchen Fällen wird gänzlich auf Massenspeicher und andere Geräte verzichtet.

2.4 Tasks

In Echtzeitsystemen werden Prozesse und Kontrollfäden (engl. *threads*) allgemein als **tasks** bezeichnet. Prozesse und Kontrollfäden unterscheiden sich in ihrer Art und Weise, wie sie im Speicher gehalten und vom Betriebssystem verwaltet werden. Prozesse haben zum Beispiel eigene Speicherbereiche für den Programmkode und die Daten, Kontrollfäden nutzen hingegen gleiche Speicherbereiche gemeinsam. Deshalb nennt man sie auch Leichtgewichtprozesse, weil durch die nicht getrennten Bereiche weniger Verwaltungsarbeit anfällt und weniger Speicher belegt wird.

Die in Ada formulierten nebenläufigen Programmeinheiten heißen auch *tasks*. Um sie näher zu bezeichnen, werden sie auch **Ada-tasks** genannt. Werden die übersetzten **Ada-tasks**, die Teile des Quellprogrammes sind, später dann ausgeführt, bilden sie die *tasks* des Systems, in dem sie laufen.

In der vorliegenden Arbeit wird der Begriff der *task* stellvertretend für Prozesse oder Kontrollfäden genannt.

Hier werden ausschließlich 1-Prozessor-Systeme betrachtet. Die zu einem 1-Prozessor-System gehörenden *tasks* müssen in einer bestimmten Art und Weise abgearbeitet werden, damit beim Benutzer der Eindruck echter Nebenläufigkeit entsteht, denn der Prozessor kann zu einem Zeitpunkt nur eine *task* bedienen. Diese scheinbare Nebenläufigkeit wird auch Pseudoparallelität genannt. Die *tasks* werden nicht vollständig und einzeln nacheinander abgearbeitet, sondern im Prinzip nur stückweise, so daß von jeder *task* nur ein Teil auf einmal abgearbeitet wird.

Für die Modellierung und Analyse von Echtzeitsystemen sind die *tasks* im wesentlichen durch ihre Kenngrößen bezeichnet. Die Grundkenngrößen von *tasks* sind:

- Aktivierungszeit. Das ist der Zeitpunkt, an dem eine *task* bereit für die Ausführung wird. Es ist zugleich der früheste Zeitpunkt, an dem eine *task* gestartet werden kann. Wenn zur Aktivierungszeit einer *task* keine höher- oder gleichpriorisierte *task* ausgeführt wird, dann wird sofort mit der Ausführung der aktivierten *task* begonnen.
- Periode. Das ist der Zeitraum zwischen zwei Aktivierungen ein und derselben *task*.
- Frist. Das ist der Zeitpunkt, bei dem die *task* beendet sein muß.
- Ausführungszeit. Das ist die Gesamtzeit, die der Prozessor zur vollständigen Ausführung einer *task* benötigt. Wir nehmen für unsere Berechnungen den schlechtesten Fall an, also die maximale Ausführungszeit (gebräuchlich: WCET, v. engl. W*orst* C*ase* E*xecution* T*ime*).
- Startzeit. Das ist der Zeitpunkt, an dem mit der Ausführung einer *task* begonnen wird.

- Beendigungszeit. Das ist der Zeitpunkt, an dem mit der Ausführung der *task* aufgehört wird.

- Antwortzeit. Das ist der Zeitraum zwischen der Aktivierungszeit und Beendigungszeit einer *task*.

Ein graphisches Beispiel soll den Sinn der Kenngrößen verdeutlichen:

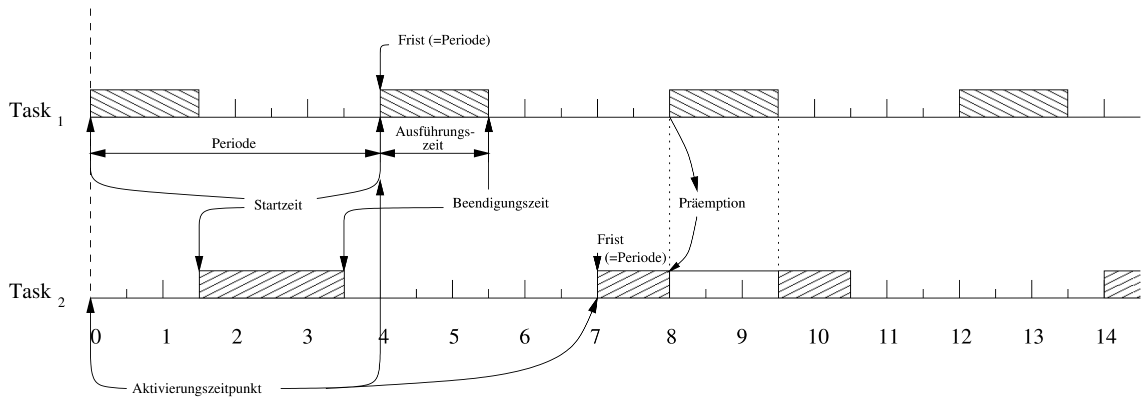

Bild 1: Kenngrößen von *tasks*

Ein weiteres Merkmal von *tasks* ist ihre **Priorität**. *Tasks* können Prioritäten haben, wenn sie in Betriebssystemen laufen, die eine Prioritätensteuerung bieten. Die Prioritäten können den *tasks* nach verschiedenen Merkmalen zugewiesen werden. Zum Beispiel kann eine *task*, die eine besonders wichtige Aufgabe erfüllen muß, etwa die Nachregulierung des Seitenruders eines Flugzeuges im Flug, die höchste aller möglichen Prioritäten erhalten. Dadurch wird sichergestellt, daß die *task* immer dann, wenn sie aktiviert wird, auch mit der Ausführung beginnen kann. Dazu **suspendiert** sie die *task*, die gerade läuft.

Wenn in dieser Arbeit *tasks* erwähnt werden, sind immer priorisierte *tasks* (*tasks* mit Prioritäten) gemeint.

Tasks lassen sich in zwei Kategorien teilen. Die *tasks* der einen werden immer nach der selben Zeitspanne aktiviert. Diese Zeitspanne heißt Periode. Die *tasks* der anderen Kategorie haben keine Perioden. Demnach gibt es **periodische** und **nichtperiodische** *tasks*.

2.4.1 Periodische *tasks*

Periodische *tasks* werden immer nach Ablauf einer Periode bereit zur Ausführung. Jede *task* hat ihre eigene Periode. Es muß sichergestellt sein, daß eine *task* ausgeführt wurde, wenn ihre Periode vorbei ist. Die maximale Ausführungszeit darf also nicht länger sein als die Periode. Es kann nicht sein, daß eine *task* erneut gestartet werden soll, wenn sie noch läuft. Es wird deshalb die Frist kleiner oder gleich der Periode gesetzt. Typische Beispiele für periodisch arbeitende *tasks* sind das wiederholte Abfragen von Sensordaten oder die periodische Systemüberwachung.

2.4.2 Nichtperiodische *tasks*

Nichtperiodische *tasks* können zu beliebigen, unvorhersehbaren Zeitpunkten aktiviert werden und sich um den Prozessor bewerben (aktiviert werden). Das zeitliche Verhalten von Systemen mit nichtperiodischen *tasks* ist schwieriger vorauszusagen als das von Systemen mit periodischen *tasks*.

In dieser Arbeit werden nur periodische *tasks* betrachtet.

2.5 Laufplanung

Bei der Laufplanung (engl. *scheduling*) wird bestimmt, wann die *tasks* aktiviert, gestartet, suspendiert oder blockiert werden. Dabei gehen die verschiedenen **Laufplanungsmethoden** verschieden vor. Ein **Laufplan** wird entsprechend der Laufplanungsmethode vom **Laufplaner** (engl. *scheduler*) erstellt.

Es gibt viele Arten von Laufplanungsmethoden, mit denen die verschiedenen Laufplaner realisiert sind. Für Laufplaner gibt es einige Kategorien.

- **verdrängend.** In Echtzeitsystemen werden in der Regel den *tasks* entsprechend ihrer Wichtigkeit Prioritäten zugewiesen. Hat eine aktivierte *task* eine höhere Priorität als die gerade laufende, dann wird die laufende *task* suspendiert (verdrängt), so daß die wichtigere sofort ausgeführt werden kann. Verdrängende Laufplaner bieten die Möglichkeit, entsprechend der zuvor vereinbarten Laufplanmethode die *tasks* zu suspendieren.

- **kooperativ.** Hierbei werden *tasks* nicht suspendiert. Jede *task* wird erst vollständig ausgeführt, bevor die nächste an die Reihe kommt.

- **statisch.** Ein Laufplaner heißt statisch, wenn er den Laufplan anhand vorgegebener, den *tasks* vor ihrer Aktivierung zugewiesener, nicht veränderbarer, also **statischer** *task*-Kenngrößen berechnet.

- **dynamisch.** Ein Laufplaner heißt dynamisch, wenn er den Laufplan für *tasks* berechnet, deren Kenngrößen auch während der Ausführung geändert werden können. Beispiel: Die *task* mit der höchsten Priorität ist mit ihrem wichtigen Teil fertig und erniedrigt ihre Priorität. Das muß vom Laufplaner sofort für die weitere Planung berücksichtigt werden, da diese *task* nun unterbrochen werden kann.

- **offline.** Ein Laufplaner arbeitet *offline*, wenn er zum Zeitpunt der Aktivierung der *tasks* den Laufplan bereits erstellt hat. Dieser Laufplan wird dann während der Laufzeit des Systems vom sogenannten *dispatcher* dazu verwendet, den Prozessor entsprechend der vorher ermittelten Zeitwerte den *tasks* zuzuweisen. Da der *offline*-Laufplaner vor der Aktivierung der *tasks* alle ihre Kenngrößen für die Laufplanerstellung kennen muß, ist er zugleich auch statisch. Einen dynamischen *offline*-Laufplaner kann es somit nicht geben.

- **online.** Ein Laufplaner arbeitet *online*, wenn er die Laufplanung während der Ausführung der *tasks* macht. Hierbei kann er auf Änderungen der *task*-Kenngrößen sofort reagieren und den Laufplan entsprechend berechnen bzw. überarbeiten. Ein *online*-Laufplaner kann dynamisch sein, aber auch statisch. Typischerweise wird man aber kaum einen *online*-Laufplaner finden, der mit statischen Mengen von *task*-Kenngrößen arbeitet.

Die letzten vier Begriffe werden in verschiedener Literatur unterschiedlich definiert oder interpretiert. Man sollte auch bei obigen Erklärungen, denen die aus [BUT97] zu Grunde liegen, vorsichtig sein. Statisch sollte nicht der Laufplaner genannt werden, sondern nur die Kenngrößen, die zu den entsprechenden *tasks* gehören. Gleiches gilt für den dynamischen Laufplaner.

Im allgemeinen gelten *offline*-Laufplaner als effektiver, denn sie haben für die Berechnung des Laufplans mehr Zeit zur Verfügung, so daß komplexere Algorithmen eingesetzt werden können. Da die *online*-Laufplaner während der Systemlaufzeit arbeiten, müssen sie ihre Entscheidungen schnell genug fällen, um nicht etwa die Ausführungen durch ihre eigene Trägheit zu blockieren.

Da im voraus nicht mit dem Auftreten einer sporadischen *task* gerechnet werden kann, weil sie zu beliebigen Zeiten oder aber gar nicht auftreten kann, ist es dann nicht möglich, einen *offline*-Laufplaner einzusetzen, der den Laufplan vor der Ausführung der *tasks* berechnet. Es muß ein *online*-Laufplaner während der Systemlaufzeit ständig kontrollieren, welche *tasks* sich gerade in welchem Status (inaktiviert, aktiviert, suspendiert, blockiert, laufend) befinden. Entsprechend ihrer Prioritäten werden diese in eine Warteschlange gebracht. Es kann jederzeit eine neue *task* hinzukommen. Sie wird dann in die Warteschlange der *tasks* mit der gleichen Priorität eingereiht, oder sie wird sofort gestartet, wenn alle anderen *tasks* niedrigere Prioritäten besitzen.

2.6 Planbarkeitsanalyse

Das Verhalten eines Echtzeitsystems muß vorhersagbar sein, um seine Zuverlässigkeit garantieren zu können. Besonders bei Echtzeitsystemen mit harten Echtzeitanforderungen ist es wichtig, die Durchführbarkeit einer Laufplanung im Voraus, d. h. vor der Ausführung der *tasks*, sicherzustellen.

Dazu müssen bei der Planung des Echtzeitsystems Kenngrößen der einzubindenden *tasks* bekannt sein. Welche Kenngrößen (vgl. S. 12) benötigt werden, hängt von der benutzten Planbarkeitsanalysemethode ab. In dieser Arbeit werden drei verschiedene Analysemethoden berücksichtigt:

- **die ratenmonotone Analyse** (**RMA**, v. engl. R*ate* M*onotonic* A*nalysis*),
- **die Analyse für die Laufplanmethode „Planen nach Fristen“** (**ED-FA**, v. engl. E*arliest* D*eadline* F*irst* A*nalysis*) und
- **die Analyse der Antwortzeiten** (**RTA**, v. engl. R*esponse* T*ime* A*nalysis*).

Sie werden in Abschnitt 3 beschrieben.

Einige Kenngrößen werden vom Entwickler festgelegt. So kann sich z. B. bei der Systemplanung ergeben, daß es nötig ist, eine bestimmte *task* jede Stunde einmal in der 59. Minute zu starten (um beipielsweise die Anwesenheit von Mitarbeitern im lokalen Netzwerk zu überprüfen).

Andere Größen können nicht durch Überlegung bestimmt werden. Die maximale Ausführungszeit etwa muß in einem komplizierten Verfahren berechnet werden. Allein die Berechnung der reinen Bearbeitungszeit der Algorithmen der *tasks* wäre eine aufwendige Sache. Es kommen aber noch umgebungsbedingte Größen hinzu. Im Abschnitt 2.6.1 werden einige dieser Größen genannt.

Durch komplexe Architektur der Hardware kommt es sicher zu den größten Schwierigkeiten bei der Vorhersage des zeitlichen Verhaltens der *tasks*. Moderne Rechenanlagen bieten zur Effektivitätssteigerung verschiedene Lösungen an. Zum Beispiel gibt es besondere Zwischenspeicher (*caches*, siehe dazu S. 19), deren

Einsatz den Programmablauf beschleunigen. Damit wird die Planbarkeitsanalyse dadurch erschwert, daß im voraus nicht bekannt ist, wann sich welche *task* im besonderen Zwischenspeicher befindet.

Schon Anfang der 70er Jahre hat man sich intensiv mit der Planbarkeitstheorie auseinandergesetzt. Besonders wichtige Ergebnisse in der Theorie der Analyse von Echtzeitsystemen mit periodischen *tasks* lieferte die Arbeit von C. L. Liu und J. W. Layland ([LL73]). Noch heute werden Echtzeitsysteme mit der von diesen beiden entwickelten Analysemethode (ratenmonotone Analyse, vgl. Abschn. 3) untersucht. Um die Planbarkeitsanalyse für periodische *tasks* zu vereinfachen, wurden einige Bedingungen an die zu analysierenden *tasks* gestellt:

1) Die Instanzen einer periodischen *task* werden in gleichen Abständen aktiviert. Der Zeitabschnitt zwischen zwei Aktivierungen heißt Periode dieser *task*.

2) Alle Instanzen einer *task* haben die gleiche maximale Ausführungszeit.

3) Alle Instanzen einer *task* haben die gleiche (relative) Frist. Diese ist gleich der Periode.

4) Alle *tasks* sind unabhängig voneinander. Z. B. nutzen sie nicht die gleichen Betriebsmittel, durch die sie blockiert werden könnten.

5) Keine *task* kann sich selbst suspendieren.

Mit **Instanz** einer *task* wird die eigentliche Arbeit bezeichnet, die ausgeführt wird. *Task* ist der abstrakte Begriff, der die Arbeit bezeichnet.

Um sichere Voraussagen über die Durchführbarkeit der Laufplanung eines Echtzeitsystems nach der von Liu und Layland beschriebenen Methode machen zu können, ist es notwendig, daß diese Bedingungen erfüllt werden.

Die darauf aufbauenden Entwicklungen auf dem Gebiet der Planbarkeitsanalyse haben diese Bedingungen dann gelockert.

An dieser Stelle werden die mathematischen Notationen für die in Abschnitt 2.4 verwendeten Begriffe eingeführt:

τ_i - *task* i;

r_i - Aktivierungszeit (engl. r*elease time*) der *task* i;

D_i - Frist (engl. d*eadline*) der *task* i;

T_i - Periode der *task* i;

C_i - maximale Ausführungszeit (engl. *computation time*) der *task* i;

s_i - Startzeit der *task* i;

f_i - Beendigungszeit (engl. *finishing time*) der *task* i;

R_i - Antwortzeit (engl. *response time*) der *task* i;

Wird ein Echtzeitsystem nach einer der bekannten Analyseverfahren untersucht, sollte vorher überprüft werden, ob das vorhandene Echtzeitbetriebssystem einen entsprechenden Laufplaner anbietet, der die *tasks* so plant, wie bei Planbarkeitsanalyse angenommen. Laufplaner und Laufplanungsanalyse müssen korrespondieren.

2.6.1 Probleme bei der Planbarkeitsanalyse

Manchmal wird gefordert, daß die Fristen der *tasks* vor den Perioden liegen. Zum Beispiel kann von einer *task* eine sehr kurze Antwortzeit verlangt werden, obwohl ihre Periode sehr lang ist. Das widerspricht Bedingung 3) (siehe oben).

Es ist auch oft unmöglich, Bedingung 4) zu akzeptieren. In vielen Systemen teilen sich verschiedene *tasks* die gleichen Betriebsmittel. Beispielsweise sind drei *tasks* mit der Berechnung einer Größe beschäftigt. Sie müßten dann auf den gleichen Speicherbereich zugreifen, um ihre Zwischenergebnisse abzulegen und andere Zwischenergebnisse zu lesen. Dabei würden sie sich beeinflussen.

Die beiden folgenden Ziele sollten bei der Planung eines Echtzeitsystems verfolgt werden:

1. Die Abarbeitung zeitkritischer Prozesse soll nicht die dauerhafte Blockierung unkritischer Prozesse nach sich ziehen.
2. Prioritätenumkehrung, auch zwischen zeitkritischen Prozessen unterschiedlicher Priorität, soll soweit wie möglich reduziert werden.

   [HORN98, 8.2.2]

Das erste Ziel kann dadurch erreicht werden, daß den *tasks* Fristen gesetzt werden. Können diese nicht eingehalten werden, wurde das Ziel nicht erreicht und die *tasks* sind nicht planbar. Die Erklärungen der Prioritätenumkehrung folgt auf Seite 23.

Cache-Speicher

Es gibt verschiedene Ursachen, die Blockaden und Verzögerungen beim Ausführen von *tasks* hervorrufen. Eine Ursache ist das Benutzen von **cache-Speichern**. Diese sind heute in fast jedem Rechner zu finden. Sie verbessern das zeitliche Verhalten der Anwendungen. Das jedoch geht auf Kosten ihrer Vorhersagbarkeit.

Cache-Speicher beeinflussen den Umgang mit den *tasks* mitunter erheblich. Es gibt zwei Arten von *cache*-Einflüssen. Man unterscheidet **innere *cache*-Einflüsse** und **äußere *cache*-Einflüsse**.

Wenn eine *task* in einen *cache*-Speicher geladen werden soll, der zu klein ist, um den gesamten Kode der *task* aufzunehmen, muß nachgeladen werden. Das geschieht typischerweise nachdem der erste Teil, d. h. der Teil, der den *cache*-Speicher füllt, abgearbeitet wurde. Dann wird der restliche Kode der *task* vom Hauptspeicher in den *cache*-Speicher kopiert. Das kostet Zeit. Diese Vorgänge nennt man innere *cache*-Einflüsse, weil sie innerhalb der Ausführung einer *tasks* ablaufen.

Befinden sich die *tasks* in einer verdrängenden Umgebung, können sie immer dann suspendiert werden, wenn sich *tasks* mit höheren Prioritäten um den Prozessor bewerben. Wird eine *task* suspendiert, um eine andere zu starten, kommt es zu einem **Kontextwechel**. Ein Kontextwechsel tritt dann ein, wenn dem Prozessor eine neue *task* zugewiesen wird.

Wird ein Kontextwechsel durch eine Suspendierung einer *task* durch eine höherpriorisierte *task* verursacht, kann das später zu einem äußeren *cache*-Einfluß führen. Zunächst wird der Kode der neuen *task* in den *cache*-Speicher geladen. Dabei kann es wiederum zu inneren Einflüssen kommen. Dann aber, wenn die Abarbeitung dieser *task* abgeschlossen ist und die vorher suspendierte weiter ausgeführt werden soll, muß der *cache*-Speicher abermals mit dem Kode der suspendierten *task* geladen werden. Das nennt man dann äußeren *cache*-Einfluß.

Es ist sicher leicht einzusehen, daß die Voraussage des zeitlichen Verhaltens solcher Situationen sehr schwierig sein kann. Es kann zu beliebigen Schachtelungen von *task*-Suspendierungen kommen, die dann jeweils innere und äußere *cache*-Einflüsse verursachen.

Es ist möglich, trotz des komplizierteren Verhaltens des Systems durch den Einsatz von *cache*-Speichern Voraussagen über die Ausführung der *tasks* zu machen. Dazu ist ein *cache*-Simulator erforderlich, der das *cache*-Verhalten in bestimmten Rechnerumgebungen berechnen kann. Ein solcher Simulator ist im Rahmen des Projektes „ADAM“ entstanden. Er basiert auf der Idee bereits existierender

Simulatoren. Er wurde außerdem für die speziellen Anforderungen der Analyse von Ada-*tasks* entwickelt.

Pipelining

Neben oben erläuterten *cache*-Speichern bieten Rechnerarchitekturen andere Mechanismen, die die Abarbeitungsgeschwindigkeit von Programmen erhöhen. Das **pipelining** ist einer davon. *Pipeling* ist eine Form der Befehls- und Datenverarbeitung durch den Prozessor. Moderne Prozessoren besitzen mehrere Ausführungseinheiten, so daß mehrere Befehle parallel ausgeführt werden können. Um parallele Ausführung zu erreichen, ist es erforderlich, die Befehle parallel zu den dazugehörigen Daten verfügbar zu haben. Dazu werden *pipelines* in die Prozessorarchitektur integriert. Sie haben mehrere Stufen. Zur Veranschaulichung seien es 5:

1. bh   Befehlsholestufe
2. bd   Befehlsdekodierungsstufe
3. oh   Operandenholestufe
4. au   Ausführungsstufe
5. sp   Speicherstufe

Für eine Erklärung des Prinzips des *pipelining* reicht es aus, von den oben aufgezählten 5 Stufen auszugehen. Es gibt Implementierungen mit z. B. 4 Stufen, wobei nicht eine weggelassen, sondern zwei zusammengefaßt werden. Man findet auch geteilte *pipelines*, bei denen Ausführungsstufen für Ganzzahlen, Fest- und Fließkommazahlen vorhanden sind.

Nehmen wir ein Beispiel mit zwei einfachen Befehlen, einer Addition und einer folgenden (unabhängigen) Subtraktion:

```
add adr1, 7    -- adr1 = adr1 + 7
sub adr2, 9    -- adr2 = adr2 - 9
```

Ein Prozessor ohne *pipelining* würde nacheinander den ersten Befehl holen, ihn dekodieren, seine beiden Operanden laden (das geschähe in einer Stufe), die Berechnung durchführen und schließlich das Ergebnis speichern. Anschließend würde er dasselbe mit dem nächsten Befehl machen. Um beide Befehle auszuführen, bräuchte der Prozessor 10 Schritte.

Ein Prozessor mit *pipelining* würde in den einzelnen Stufen parallel arbeiten. Zunächst holt er den ersten Befehl. Wenn er dann in der zweiten Stufe den Befehl dekodiert, kann er parallel dazu wieder in der ersten Stufe den zweiten Befehl holen. Im nächsten Schritt würden die Operanden für den ersten Befehl geholt und der zweite Befehl dekodiert werden. Um das bildlich zu zeigen, sei auf die Skizze der Situation in Bild 2 auf Seite 21 hingewiesen. Dort kann man die einzelnen Stufen und ihre Beziehung zueinander sehen. Es ist zu beachten, daß der Doppelkasten mit der Bezeichnung ,,paralleles Abarbeiten mit *pipelining*“ nur eine *pipeline* darstellt. Die Anordnung soll lediglich die parallele Abarbeitung der einzelnen Stufen verdeutlichen.

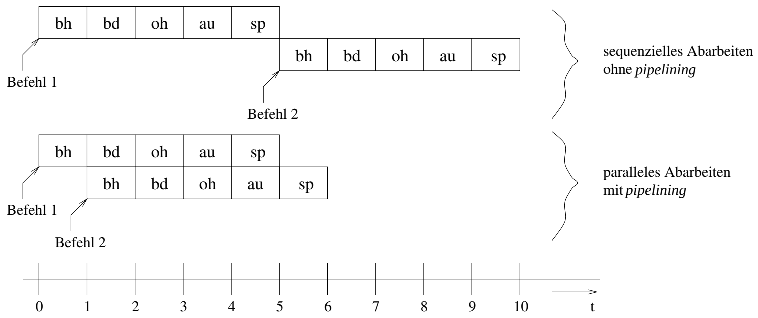

Bild 2: *pipelining*

Diese einfache Darstellung des *pipelining* stellt lediglich die Idee des *pipelining* dar. In der Praxis wird von dieser Idee ausgegangen, und so werden sehr umfangreiche und komplexe Methoden zur Verfügung gestellt. Es ist durchaus üblich, mehrere *pipelines* nebeneinander mit verschiedenen Ausführungsstufen zu haben.

Diese Methoden müssen nun in die Laufplanungsanalyse einbezogen werden, da sie das Betriebsverhalten eines Echtzeitsystems beeinflussen. Um zeitliche Vorhersagen machen zu können, muß in der Analyse ein *pipelining*-Simulator das Verhalten des Prozessors in Bezug auf seine *pipeline*-Technik immitieren. Es wird im Projekt ADAM ein *pipelining*-Simulator entwickelt, der nicht an die Architektur eines einzigen Prozessors gebunden ist, sondern der mehrere Prozessorarchitekturen berücksichtigen kann.

Betriebsmittel

Da in den meisten Echtzeitsystemen Betriebsmittel (auch Ressource, engl. *resource*) für die gemeinsame Verwendung bereitstehen, ist es unmöglich, die Bedingung 4) (2.6, S. 17) immer einzuhalten. In verdrängenden Systemen sind die *tasks* durch verschiedenhohe Prioritäten geordnet. Je wichtiger eine *task* ist, desto höher ist auch ihre Priorität. Die verschiedenen Laufplanungsalgorithmen bestimmen die Wichtigkeit der *tasks*.

Die physikalischen Komponenten von Rechenanlagen, die von den Programmen zur Datenspeicherung, -verarbeitung, -ein- und -ausgabe verwendet werden, sind Betriebsmittel. Typische Betriebsmittel sind demnach Hauptspeicher, Prozessor, Tastatur und Drucker. Auch Programme und Daten können Betriebsmittel sein. Eine Funktion etwa, auf die ein oder mehrere *tasks* zugreifen können, kann als Betriebsmittel gelten.

Um Konflikte zu minimieren, könnte der Speicher in Regionen unterteilt und jedem Rechenprozeß, bzw. jeder *task*, eine eigene Region zur Verfügung gestellt werden. Bei den Eingabekomponenten ist es schon schwieriger. Irgendwo müssen die Eingaben hergeholt werden, die von den einzelnen *tasks* bearbeitet werden sollen. Das kann nur direkt von den Eingabegeräten oder aus gemeinsam genutzten Speicherbereichen oder Dateien geschehen.

Es ist zu sehen, daß es zu Problemen kommt, wenn versucht wird, alle *tasks* völlig unabhängig voneinander halten zu wollen. Die *tasks* müssen sich also Betriebsmittel teilen. Dabei kann es verschiedene Auswirkungen auf das Gesamtverhalten des Systems haben.

Ein klassisches Beispiel ist der Versuch, einen Drucker teilen zu wollen: *Task* A beginnt damit, ihre Ergebnisse auf dem Drucker auszugeben. Nun wird *task* B mit ihrer Berechnung fertig. Auch sie wird jetzt ihre Ergebnisse an den Drucker schicken. Es kann dann beobachtet werden, wie Buchstaben und Zahlen der Ergebnisse von beiden *tasks* durcheinander auf dem Drucker ausgegeben werden.

Ähnliche Schwierigkeiten gibt es in allen Fällen, wo mehrere *tasks* auf ein und dasselbe Betriebsmittel (Ressource) zugreifen wollen. Um Konflikte zu verhindern, muß man das Betriebsmittel vor neuen Zugriffen schützen, wenn darauf bereits zugegriffen wird. Das erreicht man durch **gegenseitigen Ausschluß**.
D. h. daß ausschließlich eine *task* das angeforderte Betriebsmittel erhalten kann. Erst wenn diese *task* das Betriebsmittel freiwillig wieder abgibt, kann die nächste *task* darauf zugreifen. In einem Echtzeitsystem mit Prioritätensteuerung kann das zu einem Problem werden. Es kann zur Prioritätenumkehrung kommen.

Der (zeitliche) Abschnitt zwischen dem Zeitpunkt, wo eine *task* ein Betriebsmittel bekommt und dem Zeitpunkt, wo sie es wieder abgibt, heißt **kritischer Abschnitt**. Er ist kritisch, weil in diesem Abschnitt unvorhersehbare Blockaden eintreten können, die solche *tasks* betreffen können, die durch ihre hohe Priorität eigentlich einen ungestörten Ablauf garantiert bekommen haben.

Abschnitte, in denen eine *task* kein Betriebsmittel benutzt, heißen **unkritische Abschnitte**.

Prioritätenumkehrung

Wenn in einem verdrängenden Echtzeitsystem einige *tasks* bevorzugt werden sollen, weil ihre sofortige Abarbeitung sichergestellt sein muß, dann werden ihnen höhere Prioritäten zugewiesen als anderen. Sobald eine *task* mit einer hohen Priorität bereit wird, suspendiert (verdrängt) sie die aktuelle, wenn deren Priorität niedriger ist. Bei entsprechender Laufplanung sollte man sichergehen können, daß die hochpriorisierte *task* innerhalb ihrer Frist beendet ist. Das Phänomen der Prioritätenumkehrung kann das verhindern. Nehmen wir ein Beispiel:

Gegeben seien drei *tasks* τ_1, τ_2 und τ_3. τ_1 habe die höchste Priorität, τ_3 die niedrigste. τ_3 läuft und fordert Betriebsmittel R an. Da es frei ist, kann τ_3 es erhalten. Nun wird τ_1 bereit. Da ihre Priorität die höchste ist, wird τ_3 suspendiert und τ_1 läuft los. Jetzt möchte auch τ_1 R haben, wird aber blockiert, da es schon von τ_3 gehalten wird. Inzwischen wurde τ_2 bereit, so daß sie sofort ausgeführt wird. τ_3 bleibt weiterhin suspendiert. Da es keine bereite *task* außer τ_2 gibt, kann diese ungestört und unbegrenzt laufen. Man sagt, τ_2 blockiert τ_1 indirekt. Die eigentliche Umkehrung der Prioritäten ist der scheinbare Tausch der Prioritäten von τ_1 und τ_3.

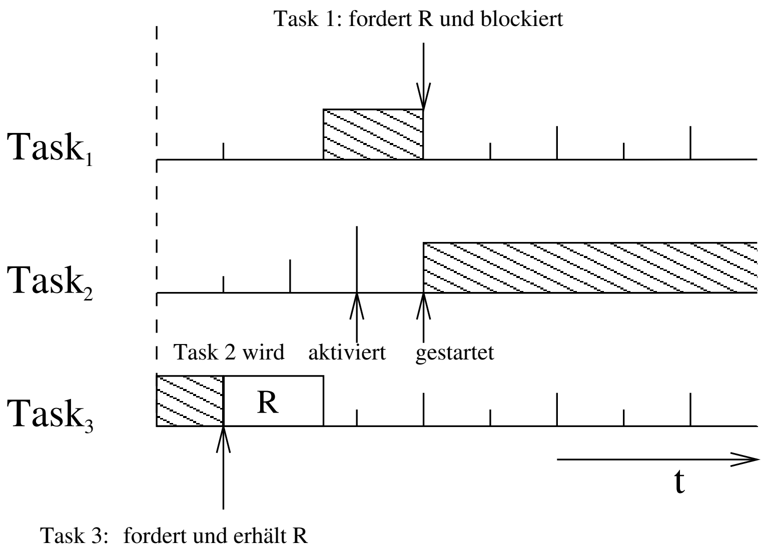

Bild 3: Prioritätenumkehrung

Dieses Problem sollte vermieden werden. Eine Planbarkeitsanalyse wird durch die Prioritätenumkehrung wesentlich erschwert. Wie diese Situation vermieden werden kann, wird in Abschnitt 3.3 erklärt.

Blockierzeiten

Wenn eine *task* aktiviert wird, dann **suspendiert** sie eine momentan laufende niederpriorisierte *task*. Wenn aber eine *task* ein Betriebsmittel benutzt, dann **blockiert** sie eine höherpriorisierte *task*, wenn diese auch das Betriebsmittel anfordert.

Die Zeit, in der die höherpriorisierte *task* am Betriebsmittel blockiert ist, heißt **Blockierzeit**. Das ist eine Möglichkeit, bei der höherpriorisierte *tasks* auf niederpriorisierte *tasks* warten müssen. Die andere, bei der indirekt blockiert wird, wurde im letzten Abschnitt erklärt.

Für eine Planbarkeitsanalyse sind mehrere Kenngrößen der *tasks* (siehe S. 2.4) notwendig. Bei der Nutzung von Betriebsmitteln sind außerdem die Blockierzeiten der *tasks* wichtig. Diese werden in komplizierten Verfahren errechnet. Abschnitt 3.3 erläutert die Berechnung der Blockierzeiten durch unterschiedliche Verfahren.

2.7 Die Programmiersprache Ada

Ada ist eine Programmiersprache, die im Auftrag des amerikanischen Verteidigungsministeriums entwickelt wurde. Die jahrelange Entwicklung brachte eine Programmiersprache hervor, die in der Hauptsache für große, komplexe, zuverlässige Systeme oder Echtzeitsysteme gedacht ist. Nach einer größeren Revision existiert nun die Version von 1995, **Ada95**, neben der älteren Version **Ada83**.

Schon bei **Ada83** gab es Möglichkeiten zur Erzeugung von *tasks* sowie deren Synchronisation. Zudem konnten *tasks* Prioritäten zugewiesen werden. Damit war eine verdrängende Laufplanung durchführbar und Vorraussetzungen für einen Echtzeitbetrieb gegeben.

Rendezvous können zur unmittelbaren Kommunikation und Synchronisation von *tasks* eingesetzt werden. Dazu besitzen die beteiligten *tasks* Mechanismen, die die Synchronisation an einer bestimmten Stelle, dem Synchronisationspunkt (engl. *entry*), gewährleisten. Ein Synchronisationspunkt befindet sich innerhalb einer *task* und wird von einer anderen *task* aufgerufen. An dieser Stelle treffen sich beide, nachdem die früher angekommene *task* dort auf das Eintreffen der anderen gewartet hat. Nach Ausführung der Anweisungen, die mit dem Synchronisationspunkt verbunden sind, trennen sich beide *tasks* wieder und setzten unabhängig voneinander ihre Arbeit fort.

Mit der Spracherweiterung in **Ada95** kamen einige wichtige Veränderungen in Bezug auf die nebenläufige Programmierung. So können viele Synchronisationsprobleme nun besser durch geschützte Objekte realisiert werden. Geschützte Objekte gab es bei **Ada83** nicht. Natürlich ist der Einsatz von *Rendezvous* weiterhin möglich.

Die Verwendung geschützter Objekte ist der mittelbare Weg zur Interaktion zwischen *tasks*. Es können mit ihnen Betriebsmittel realisiert werden, auf die mehrere *tasks* zugreifen können. Ada 95 bietet für die geschützten Objekte Mechanismen zur Synchronisation von *tasks*. Hierbei befinden sich die Synchronisationspunkte (*entries*) nicht in den zu synchronisierenden, zur Kommunikation aufgeforderten *tasks* selbst, sondern in den geschützten Objekten. Diese sind Konstrukte, die den beteiligten *tasks* zur gemeinsamen Benutzung zur Verfügung gestellt werden können. Neben den Synchronisationspunkten (*entries*) können die geschützten Objekte auch Funktionen und Prozeduren enthalten. Die Aufrufe der in einem geschützten Objekt enthaltenen Unterprogramme geschehen unter gegenseitigem Ausschluß. Somit kann es zu keinen Behinderungen durch andere Unterprogramme desselben geschützten Objekts kommen. Geschützte Objekte sind passiv. Werden ihre Unterprogramme nicht aufgerufen, gibt es keinen Einfluß auf das Gesamtverhalten.

Eine andere Neuerung ist das Erlauben von dynamischen Prioritätenzuweisungen. Dadurch wird es möglich, zwischenzeitlich einer *task* eine hohe Priorität zu geben, damit diese dann alsbald ausgeführt werden kann bzw. ihre laufende Ausführung sichergestellt ist. Nachdem die für diese *task* wichtige Aufgabe, wie das Benutzen eines Betriebsmittels, erfüllt ist, kann sie mit normaler Priorität fortgesetzt werden.

3 Planbarkeitsanalysemethoden

Planbarkeitsanalysen sind verschieden für nichtperiodische *tasks* und für periodische *tasks*. Eine Menge nichtperiodischer *tasks* enthält mindestens eine *task*, die zu einem nicht vorherbestimmbaren Zeitpunkt aktiviert wird. Selbst wenn mit nur einer einzigen Aktivierung einer nichtperiodischen *task* zu rechnen ist, kann kein *offline*- Laufplaner mehr eingesetzt werden. Es ist aber möglich, *online*-Laufplaner für die Laufplanung periodischer *tasks* zu benutzen.

In diesem Abschnitt werden drei Planbarkeitsanalysemethoden für periodische *tasks* vorgestellt. Diese Methoden werden auch von dem Planbarkeitsanalysator eingesetzt. Es handelt sich um grundlegende Methoden. Sie tauchen in der Literatur immer wieder auf. Die drei Methoden lassen sich erweitern.

Zunächst werden wichtige Begriffe erklärt.

3.1 Prozessorbelastung

Die weiter unten vorgestellten Analysemethoden *RMA* und *EDFA* beruhen auf der Untersuchung der **Prozessorbelastung** der zu prüfenden *tasks*. Übersteigt die Prozessorbelastung, die durch die *task*-Menge verursacht wird, einen bestimmten Wert, dann können diese *tasks* nicht ausgeführt werden, ohne daß sie folgenbehaftete Verzögerungen hervorrufen würden.

Die Prozessorbelastung U_i einer *task* τ_i ist das Verhältnis von der maximalen Ausführungszeit C_i zu der Periode T_i:

$$U_i = \frac{C_i}{T_i}.$$

Typischerweise werden bei der Analyse periodischer *tasks* alle *tasks*, die in dem zu planenden System enthalten sein sollen, zusammen betrachtet. Dabei ergibt sich eine Prozessorbelastung U_{ges} für die gesamte Menge der *tasks* $\tau_1 \ldots \tau_n$ von

$$U_{ges} = \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

Der Prozessor ist ausgelastet, wenn seine Belastung 100% beträgt. Dies entspricht einem Prozessorbelastungswert von 1,0. Diese Maximalbelastung soll hier U_{max} heißen. Liegt der Wert U_i einer *task* τ_i über U_{max}, dann ist τ_i garantiert nicht planbar. Liegt der Wert U_{ges} einer Menge von *tasks* $\tau_1 ... \tau_n$ über U_{max}, dann ist die gesamte Menge nicht planbar.

Daraus ist nicht zu schließen, daß bei beliebigen Werten für $U_{ges} \leq U_{max}$ die Menge der *tasks* garantiert planbar ist. Es gibt aber einen Grenzwert, wobei die Planbarkeit der untersuchten Menge garantiert werden kann, wenn der Wert U_{ges} der Menge darunter bleibt. Dieser Grenzwert heißt hier **Prozessorbelastungsgrenze** U_{grenz}. Der Wert der Prozessorbelastungsgrenze hängt von der Analysemethode ab, mit der die *tasks* untersucht werden.

3.2 Kritische Zeitzone

Die Antwortzeit einer *task* ist die Zeitspanne zwischen der Aktivierungszeit der *task* und der Beendigungszeit derselben. In dieser Antwortzeit sind die Ausführungszeit und die Wartezeit, die durch Suspendierungen entstehen, enthalten. Für die höchstpriorisierte *task* ist die Antwortzeit gleich der Ausführungszeit, weil sie nicht suspendiert werden kann. (Es sei an dieser Stelle angenommen, daß keine Blockierzeiten durch Betriebsmittelnutzung mit einfließen.)

Die maximale Antwortzeit einer *task* heißt **kritische Zeitzone**. Die kritische Zeitzone einer *task* wird erreicht, wenn die *task* so häufig wie möglich von höherpriorisierten *tasks* suspendiert wird. Eine niederpriorisierte kann durch eine höherpriorisierte *task* umso öfter suspendiert werden, je früher sie aktiviert wird. Sie kann am häufigsten suspendiert werden, wenn ihre Aktivierungszeit gleich der Aktivierungszeit der höherpriorisierten *task* ist. Dieser Zeitpunkt, wo die Aktivierung einer *task* deren kritische Zeitzone erzeugt, heißt **kritischer Moment**. Bild 4 zeigt eine solche Situation am Beispiel.

In beiden Fällen hat *task* 2 eine Periode von 12. Im Teil a) wird *task* 2 jedoch eine Einheit später aktiviert als in Teil b). Es kommt bei b) zu zwei Suspendierungen innerhalb der ersten 13 Zeiteinheiten und bei a) zu keiner.

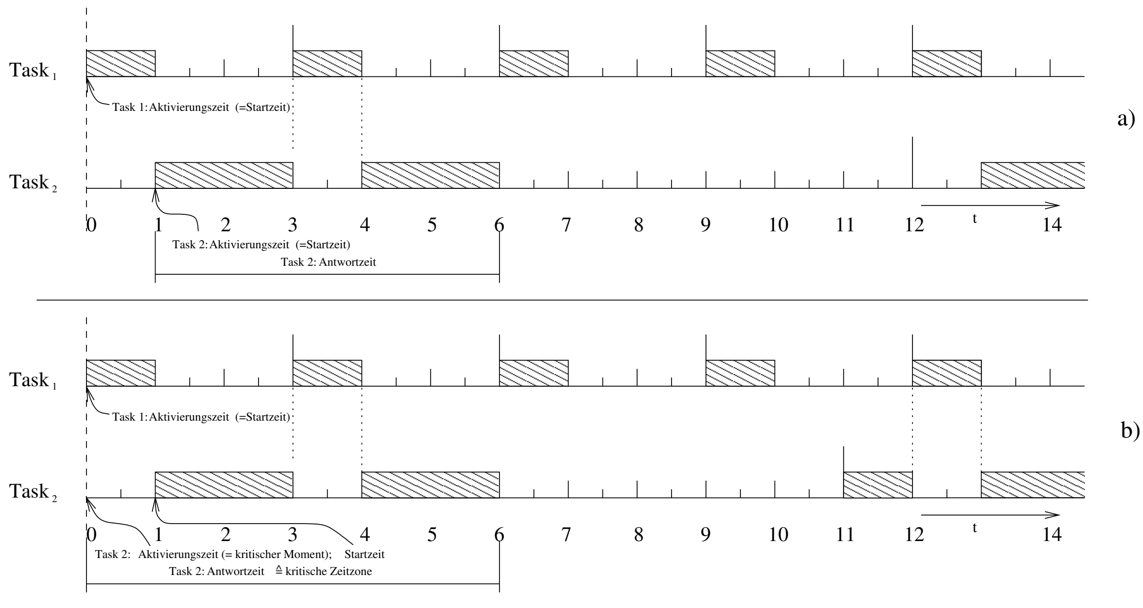

Bild 4: a) spätere Aktivierung von *task* 2, $R_2 = 5$; b) frühere Aktivierung von *task* 2 (zum kritischen Moment), $R_2 = 6$

Für eine *task*-Menge entspricht die kritische Zeitzone der Antwortzeit der niedrichstpriorisierten *task*, denn durch die höheren Prioritäten der anderen *tasks* ist sichergestellt, daß diese zuerst ausgeführt und beendet werden.

Die kritische Zeitzone wurde bereits in [LL73] präsentiert. Liu und Layland bewiesen ferner, daß keine *task* der zu untersuchenden Menge je ihre Frist verpassen wird, wenn auch innerhalb der kritischen Zeitzone keine Frist verpaßt wurde. Erstellt man eine Simulation einer Laufplanung für eine *task*-Menge innerhalb der kritischen Zeitzone, kann man damit die Planbarkeit der Menge überprüfen. Werden alle Fristen eingehalten, ist die Menge planbar.

Der Planbarkeitsanalysator berechnet die kritische Zeitzone für eine zu untersuchende *task*-Menge. Dabei simuliert er die Laufplanung nach der ratenmonotonen Methode. Siehe dazu Abschnitt 8.1.

3.3 Zugriff auf Betriebsmittel

In vielen Fällen wird es vorkommen, daß in einem System sich die *tasks* Betriebsmittel teilen müssen. Dann wird es schwieriger, das Verhalten der *tasks* vorauszusagen. Auf den Seiten 21ff. wurde bereits auf die Probleme im Umgang mit Betriebsmitteln eingegangen.

Um das Problem der Prioritätenumkehrung zu umgehen, wurden Verfahren entwickelt, durch die die *tasks* bei der Benutzung der Betriebsmittel ein verändertes Verhalten zeigen.

Diese Verfahren werden **Betriebsmittelzugriffsprotokolle** genannt. Zwei dieser Protokolle werden vom Planbarkeitsanalysator berücksichtigt.

Protokoll des Prioritätenerbens

Dieses Betriebsmittelzugriffprotokoll wurde in [SRL90] vorgestellt. Das Prinzip ist einfach. Ein Beispiel soll es erklären:

Gegeben seien 3 *tasks* τ_1, τ_2, τ_3 mit den Prioritäten $P_1 = 3$ (hoch), $P_2 = 2$ (mittel), $P_3 = 1$ (niedrig). τ_1 und τ_3 greifen auf Betriebsmittel R zu.

Ablauf:

1. τ_3 fordert und erhält R.
2. τ_1 startet und suspendiert τ_3.

3. τ_1 fordert R an und blockiert an R.
4. τ_3 erbt P_1 und setzt die Ausführung im kritischen Abschnitt fort.
5. τ_2 wird bereit, kann aber nicht starten, weil τ_3 mit P_1 läuft.
6. τ_3 beendet die Ausführung im kritischen Abschnitt und erhält P_3 zurück.
7. τ_1 suspendiert τ_3.
8. τ_1 erhält R, läuft bis zum Ende des kritischen Abschnitts und dann weiter bis Ende ihrer Ausführungszeit.
9. τ_2 startet, läuft und beendet.
10. τ_3 läuft bis zum Ende.

   ⋮

Damit wird verhindert, daß τ_2 τ_3 im kritischen Abschnitt suspendiert und somit τ_1 im blockierten Zustand hält, bis τ_2 mit der Ausführung fertig ist. Bild 5 zeigt den Ablauf graphisch.

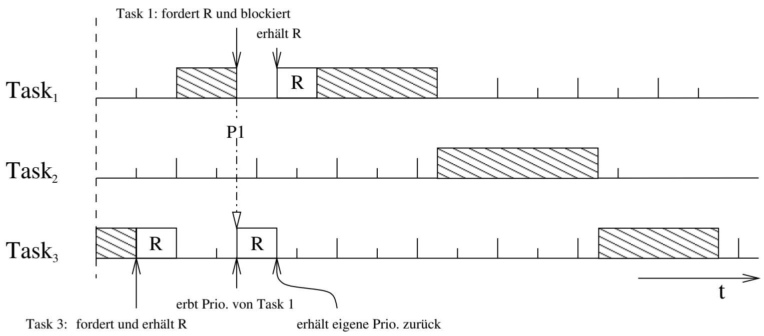

Bild 5: Prioritätenerben

Welche *tasks* andere *tasks* blockieren können, ist auf Seite 24 erläutert.

Hier ist ein Beispiel aus aus [BUT97]. Die zugehörige Tabelle steht auf Seite 30.

In der Tabelle stehen die längsten kritischen Abschnitte, die je *task* mit den Betriebsmitteln zustande kommen. Angenommen, hier benutzt τ_2 Betriebsmittel R_2 genau dreimal, d. h. τ_2 tritt dreimal in einen kritischen Abschnitt mit R_2. Die Ausführungszeiten der kritischen Abschnitte seien 6, 4 und 9. Der längste kritische Abschnitt von τ_2 mit R_2 ist dann der mit der Ausführungszeit von 9.

	R1 $P_C = P1$	R2 $P_C = P1$	R3 $P_C = P2$
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

Bild 6: Tabelle für die Errechnung der Blockierzeiten

Die Blockierzeit einer *task* τ_i unter Berücksichtigung des Protokolls des Prioritätenerbens (engl. *Priority Inheritance Protocol*) errechnet sich, indem zunächst zwei Werte ermittelt werden, die anschließend miteinander verglichen werden. Der niedrigere Wert ist die Blockierzeit der *task*.

Wert 1 (Blockierzeit, entstehend durch alle *tasks*, die τ_i blockieren können):

- Betrachte alle *tasks*, die τ_i blockieren können!
- Ermittle den längsten kritischen Abschnitt jeder dieser *tasks*!
- Summiere diese Werte!

Wert 2 (Blockierzeit, entstehend durch alle Betriebsmittel, an denen τ_i blockieren kann):

- Betrachte alle Betriebsmittel, an denen τ_i blockieren kann, die τ_i also anfordern wird, aber nur in den Zeilen der niederpriorisierten *tasks*!
- Ermittle den längsten kritischen Abschnitt jedes dieser Betriebsmittel!
- Summiere diese Werte!

• Vergleiche Wert 1 mit Wert 2! Der kleinere ist die maximale Blockierzeit der untersuchten *task*.

Für obige Tabelle sind die Werte und Summen:

task 1:
Wert $1 = 9 + 8 + 6 = 23$
Wert $2 = 8 + 9 = 17$
task 2:
Wert $1 = 8 + 6 = 14$

Wert 2 = 8 + 7 + 4 = 19

task 3:

Wert 1 = 6

Wert 2 = 6 + 5 + 4 = 15

task 4:

Wert 1 = Wert 2 = 0

Nachdem jeweils die Werte 1 und 2 verglichen worden sind, ergeben sich folgende Blockierzeiten für die *tasks*: $B_1 = 17$, $B_2 = 14$, $B_3 = 6$ und $B_4 = 0$. B_4 hat keine Blockierzeit, weil es keine niederpriorisierten *tasks* gibt, die B_4 blockieren könnten.

Das Testbeispiel auf Seite 64ff. rechnet mit dem gleichen Beispiel, wie es hier gezeigt wurde. Es ist [BUT97] entnommen.

Protokoll der Grenzprioritäten

Auch dieses Protokoll wurde in [SRL90] vorgestellt. Es ist eine Erweiterung des Protokolls des Prioritätenerbens. Es hat den Vorteil, daß es Versperrungen (engl. *deadlocks*) verhindert.

Bei dem Protokoll der Grenzprioritäten (engl. *Priority Ceiling Protocol*) wird jedem Betriebsmittel eine Grenzpriorität zugewiesen. Diese Priorität entspricht der der höchstpriorisierten *task*, die das Betriebsmittel anfordert. Wird irgendein Betriebsmittel von einer beliebigpriorisierten *task* benutzt, dann kann eine andere *task* nur dann auch irgendein Betriebsmittel erhalten, wenn die Priorität der *task*, die das Betriebsmittel anfordert, höher ist als die Grenzprioritäten aller momentan benutzten Betriebsmittel. Wie beim Prioritätenerben wird auch hier die Priorität einer höherpriorisierten *task* an die *task* vererbt, die das Betriebsmittel benutzt, das die höherpriorisierte *task* anfordert.

Ein Beispiel soll den Sinn verdeutlichen:

Gegeben seien 3 *tasks* τ_1, τ_2, τ_3 mit den Prioritäten $P_1 = 3$ (hoch), $P_2 = 2$ (mittel), $P_3 = 1$ (niedrig). τ_1 und τ_3 greifen auf die Betriebsmittel R_1 und R_2 zu.

Ablauf:

1. τ_3 fordert und erhält R_1.
2. τ_1 startet und suspendiert τ_3.

3. τ_1 fordert R_2 an und blockiert an R_2, weil R_2 die Grenzpriorität entsprechend der höchstpriorisierten *task* hat, die auf R_2 zugreifen kann, und das ist τ_1 selbst.

4. τ_3 erbt P_1 und setzt die Ausführung im kritischen Abschnitt mit R_1 fort.

5. τ_2 wird bereit, kann aber nicht starten, weil τ_3 mit P_1 läuft.

6. τ_3 fordert R_2 an und erhält es.

7. τ_3 führt den kritischen Abschnitt mit R_2 zu Ende aus und gibt R_2 wieder frei.

8. τ_3 führt den kritischen Abschnitt mit R_1 zu Ende aus und gibt R_1 wieder frei.

9. τ_3 erhält P_3 zurück.

10. τ_1 suspendiert τ_3.

11. τ_1 erhält R_2 und läuft ein Stück.

12. τ_1 fordert und erhält R_1.

13. τ_1 läuft bis zum Ende beider kritischenAbschnitte und dann weiter bis Ende ihrer Ausführungszeit.

14. τ_2 startet, läuft und beendet

15. τ_3 läuft bis zum Ende
    $\vdots$

Mit dem Protokoll des Prioritätenerbens würde bei 3. τ_1 R_2 erhalten. Beim Schritt 6. würde dann τ_3 R_2 anfordern und blockieren. Dann würde τ_1 R_1 anfordern und blockieren. τ_1 und τ_3 wären damit für ewig versperrt. Das Protokoll der Grenzprioritäten verhindert das.

Für die Analyse der Planbarkeit einer Menge, bei der dieses Betriebsmittelzugriffsprotokoll angewendet werden soll, wird die maximale Blockierzeit jeder *task* τ_i auf folgende Weise ermittelt:

- Betrachte alle *tasks*, die τ_i blockieren können!

- Betrachte diejenigen kritischen Abschnitte dieser *tasks*, die durch Benutzung von den Betriebsmitteln zustandekommen, durch die τ_i blockiert werden kann. Diese Betriebsmittel haben eine Grenzpriorität, die größer oder gleich der Priorität von τ_i ist!

- Nimm den größten Wert dieser ermittelten kritischen Abschnitte!

Somit hat man die Blockierzeit für τ_i. Für die *task* mit der niedrigsten Priorität braucht man dieses Verfahren nicht anwenden (s. S. 24).

Wenn dann die Blockierzeiten aller *tasks* vorliegen, können sie in die erweiterten Formeln der im folgenden beschriebenen Planbarkeitsanalysemethoden eingesetzt werden.

3.4 Ratenmonotone Analyse (RMA)

Der **ratenmonotonen Analyse** (engl. rate monotonic analysis, RMA) liegt die **ratenmonotone Laufplanung** zugrunde. Bei dieser Laufplanungsmethode werden den zu planenden *tasks* entsprechend ihrer Raten die Prioritäten zugewiesen. Dabei haben *tasks* mit kurzen Perioden hohe Raten und *tasks* mit langen Perioden niedrige Raten. Die *task* mit der höchsten Rate (mit der kürzesten Periode) bekommt die höchste Priorität. Den *tasks* mit absteigend niedrigeren Raten werden monoton absteigende Prioritäten zugewiesen. Nach dieser Zuweisungsmethode priorisierte *tasks* werden mit der ratenmonotonen Laufplanmethode geplant.

In [BUT97, S. 82ff.] und [LL73] wurde die Optimalität der ratenmonotonen Laufplanmethode bewiesen. ,,Die ratenmonotone Laufplanmethode ist optimal“ bedeutet, es gibt keine andere Laufplanmethode, die eine Menge von *tasks* planen kann, wenn die ratenmonotone Methode diese Menge nicht planen kann. Das beschränkt sich auf periodische *tasks* mit unveränderbaren Prioritäten und Fristen, die gleich den Perioden sind.

Ein Vorteil gegenüber anderen Laufplanmethoden ist der, daß sich die Durchführbarkeit einer Laufplanung einfach errechnen läßt. Eine Laufplanung für eine *task*-Menge ist durchführbar, wenn an keiner Stelle eine Instanz einer *task* ihre Frist verpaßt.

Wie im Abschnitt 3.1 erwähnt, ist eine Laufplanung einer Menge von *tasks* dann garantiert nicht durchführbar, wenn die Prozessorbelastung dieser Menge den Wert 1,0 übersteigt.

Die ratenmonotone Analyse (im Folgenden *RMA*) kann garantieren, daß eine

Menge von *tasks* planbar und ausführbar ist, wenn der Wert der Prozessorbelastung kleiner als der Wert der *RMA*-bezogenen Prozessorbelastungsgrenze ist. Formal dargestellt heißt das:

$$U_{ges} \leq U_{grenz}.$$

Bei $U_{grenz} \leq U_{ges} \leq U_{max}$ kann durch *RMA* nicht garantiert werden, daß die *task*-Menge planbar ist. Es wird (aus Sicherheitsgründen) in dieser Arbeit angenommen (bestimmt), daß eine *task*-Menge nicht planbar ist, wenn $U_{ges} > U_{grenz}$. In der Wirklichkeit kann es aber durchaus sein, daß die Menge trotzdem planbar ist.

Die Prozessorbelastungsgrenze U_{grenz} für *RMA* ist von Liu und Layland hergeleitet worden (in [LL73]). Auch [BUT97] zeigt eine Herleitung, zunächst für zwei *tasks* (S. 85–89), dann für n *tasks* (S. 89–92). Diese Grenze U_{grenz} liegt für n *tasks* bei

$$n(2^{\frac{1}{n}} - 1).$$

In dieser Arbeit wird keine Herleitung dieser Prozessorbelastungsgrenze gemacht.

Damit garantiert werden kann, daß eine *task*-Menge planbar ist, muß sie folgende Ungleichung erfüllen:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1).$$

U_{grenz} ist umso kleiner, umso größer die Zahl der *tasks*, n, ist. Für die ersten n ($n = 2...5$) ergeben sich folgende Werte für U_{grenz}:

$$\begin{aligned}
U_{grenz(1)} &= 1(2^{\frac{1}{1}} - 1) \\
&= 1(2-1) \\
&= 1 \cdot 1 \\
&= 1 \\
U_{grenz(2)} &= 2(2^{\frac{1}{2}} - 1) \\
&= 2(\sqrt{2} - 1) \\
&\approx 2(1,414 - 1) \\
&\approx 2 \cdot 0,414 \\
&\approx 0,828 \\
U_{grenz(4)} &\approx 0,757 \\
U_{grenz(6)} &\approx 0,735 \\
U_{grenz(8)} &\approx 0,724 \\
U_{grenz(10)} &\approx 0,718
\end{aligned}$$

Schließlich konvergiert U_{grenz} gegen $\ln 2 \approx 0,693$:

$$\lim_{n \to \infty} U_{grenz} = \ln 2.$$

Daraufhin kann man die Aussage machen, daß alle *task*-Mengen, die einen Prozessorbelastungswert nicht größer als 0,693 (genauer: $\ln 2$) haben, planbar sind, wenn sie nach der ratenmonotonen Laufplanungsmethode geplant werden.

Erweiterung für Betriebsmittel

Wenn *tasks* Betriebsmittel benutzen, dann kann die oben vorgestellte Formel nicht benutzt werden, um die Planbarkeit einer *task*-Menge zu bestimmen. Im Falle der Betriebsmittelnutzung kommen Blockierzeiten für die einzelnen *tasks* hinzu, die in obiger Formel nicht berücksichtigt werden.

Die Erweiterung dieser Formel lautet:

$$\forall i, 1 \leq i \leq n, \; \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i(2^{\frac{1}{i}} - 1).$$

Hierbei wird für jede *task* einzeln überprüft, ob durch ihre Ausführung nicht die Prozessorbelastungsgrenze überschritten wird.

3.5 Analyse für das „Planen nach Fristen" (EDFA)

Diese Analysemethode untersucht eine Menge von *tasks* daraufhin, ob sie mit der Laufplanmethode **„Planen nach Fristen"** (engl. *Earliest Deadline First*, EDF [früheste Frist zuerst]; EDFA – EDF Analysis) ausführbar ist.

Bei der Laufplanmethode „Planen nach Fristen" handelt es sich um einen *online*-Laufplaner, bei dem die Planungsentscheidungen während der Systemlaufzeit getroffen werden. Die auszuführenden *tasks* haben dabei keine festen Prioritäten. Sie bekommen die Prioritäten dynamisch („vor Ort") zugewiesen.

Wenn eine *task* aktiviert wird, sortiert der Laufplaner diese *task* in eine Warteschlange ein derart, daß die *tasks* mit früheren Fristen vor denen mit späteren Fristen stehen. Die *task* mit der frühesten Frist wird ausgeführt. Sie wird dann suspendiert, wenn eine *tasks* aktiviert wird, die eine frühere Frist hat. Ansonsten wird die momentan laufende *task* bis zu Ende ausgeführt. Danach wird die nächste *task*, jene, die sich am Kopf der Warteschlange befindet, gestartet.

Diese Laufplanmethode erlaubt einer Menge von *tasks*, den Prozessor bis zu 100% zu belasten, ohne daß eine *task* ihre Frist verpaßt. Das heißt, daß bei *EDF* die Prozessorbelastungsgrenze U_{grenz} bei 1 liegt. Somit erhalten wir die Bedingung für die Planbarkeit einer *task*-Menge:

$$U_{ges} \leq 1.$$

Um eine *task*-Menge zu analysieren, die nach dieser Laufplanmethode geplant werden soll, wird die Gesamtprozessorbelastung der Menge berechnet und mit dem Grenzwert verglichen. Belastet die *task*-Menge den Prozessor nicht mehr als 100%, dann wird keine *task* je ihre Frist verpassen und die Menge ist ausführbar. Die Vorgehensweise entspricht der von *RMA*. Die Formel sieht demnach so aus:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

3.6 Analyse der Antwortzeiten (RTA)

Bei den beiden oben beschriebenen Analyseverfahren wird davon ausgegangen, daß die Fristen der *tasks* gleich ihren Perioden seien. Das ist in der Praxis nicht immer wünschenswert.

Die **Analyse der Antwortzeiten** (engl. *Response Time Analysis*, RTA) kann auch für *tasks* mit Fristen, die vor den Perioden liegen, garantieren, daß sie planbar sind. Dabei ist der Analysealgorithmus nicht ganz trivial. Es muß iterativ der Wert der berechneten Antwortzeit aktualisiert werden, bis er sich nicht mehr ändert.

Die Antwortzeit einer *task* besteht sowohl aus ihrer eigenen Ausführungszeit als auch aus den Ausführungszeiten höherpriorisierter *tasks*. Die höherpriorisierten werden die niederpriorisierte *task* solange und so oft suspendieren, wie sie zwischen Aktivierungszeit und Frist der niederpriorisierten *task* ausgeführt werden. Bild 7 zeigt den Sachverhalt.

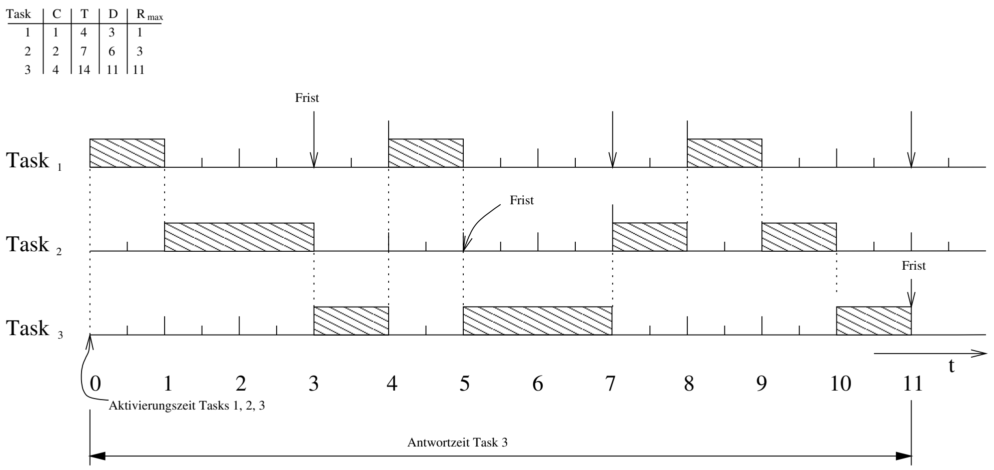

Bild 7: maximale Antwortzeit

Die Antwortzeit einer *task* τ_i könnte so berechnet werden:

$$R_i = C_i + \sum_{j=1}^{i-1} \lceil \frac{D_i}{T_j} \rceil C_j .$$

Das wäre aber etwas zu ungenau, denn hier wird damit gerechnet, daß höherpriorisierte *tasks* tatsächlich so oft wie $\lceil \frac{D_i}{T_j} \rceil$ Male τ_i unterbrechen. Das muß nicht sein, denn τ_i könnte früher beendet sein, noch bevor es zu einer weiteren Suspendierung kommt. Dazu soll angenommen werden, *task* 3 aus Bild 7 habe eine Ausführungszeit von 3, anstatt von 4. In diesem Falle würde *task* 3 an Stelle 7 beendet sein, und die Antwortzeit wäre um 4 Einheiten kürzer. Die Formel von oben nimmt an, daß τ_3 von τ_1 $\lceil \frac{D_i}{T_j} \rceil$ mal unterbrochen wird für insgesamt $\lceil \frac{D_i}{T_j} \rceil C_j$ Einheiten. Das entspricht 3 Unterbrechungen und einer Dauer von $3 \cdot 1 = 3$ Einheiten. Wäre τ_3 schon bei 7 beendet, könnte sie nur zweimal von τ_1 suspendiert werden.

Eine genauere Berechnung der Antwortzeit ist eine iterative Formel. Diese ist im Prinzip eine formale Umsetzung der Berechnung der kritischen Zeitzone (siehe Abschnitt 3.2). Diese Formel wird algorithmisch gelöst. Dabei wird solange das Ergebnis der letzten Iteration in die rechte Seite der Formel eingesetzt, bis die Gleichung erfüllt ist oder das Ergebnis den Wert der Frist der untersuchten *task* übersteigt.

$$R_i^{k+1} = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^k}{T_j} \rceil C_j.$$

Wird für eine Menge von *tasks* durch *RTA* garantiert, daß sie planbar ist, muß auch eine entsprechende Laufplanmethode angewendet werden. Die ratenmonotone Laufplanmethode ist hier nicht mehr anwendbar, da diese nur für *tasks* mit Fristen, die den Perioden entsprechen, einsetzbar ist. Die *EDF*- Methode scheidet auch aus, da sie den *tasks* die Prioritäten dynamisch zuweist, die *RTA*-Methode jedoch nur statisch priorisierte *tasks* untersucht.

Die hier einzusetzende Laufplanmethode heißt **fristenmonotone Laufplanmethode**. Sie weist die Prioritäten entsprechend den Fristen zu. Dabei bekommen die *tasks* mit kürzeren Fristen höhere Prioritäten.

Setzt man die Fristen den Perioden gleich, dann kann die Analyse der Antwortzeiten auch für die Analyse von *tasks*, die mit der ratenmonotonen Laufplanmethode zu planen sind, untersucht werden. Das wird dann gemacht, wenn U_{ges} einer Menge den Wert U_{grenz} übersteigt, aber noch unter U_{max} bleibt. Ist die Menge dann laut *RTA* planbar, kann sie ratenmonoton geplant werden.

Erweiterung für Betriebsmittel

Sollen von den *tasks* Betriebsmittel benutzt werden, dann können sie blockiert werden. Wie lange sie blockiert werden, hängt unter anderem vom Betriebsmittelzugriffsprotokoll ab (siehe dazu S. 28). Die Formel zur Berechnung der Antwortzeiten muß so verändert werden, daß sie auch die Blockierzeiten berücksichtigt.

Die Antwortzeiten werden unter Einbeziehung der Blockierzeiten mit folgender Formel errechnet:

$$R_i^{k+1} = C_i + B_i \sum_{j=1}^{i-1} \lceil \frac{R_i^k}{T_j} \rceil C_j.$$

Es wird sonst dieselbe Vorgehensweise wie bei der Berechnung der Antwortzeiten ohne Betriebsmittelzugriff angewendet.

4 Eingabe für den Planbarkeitsanalysator

Um zu überprüfen, ob sich eine Menge von *tasks* sicher planen läßt, ist es grundsätzlich notwendig, die Kenngrößen der einzelnen *tasks* (siehe 2.4, S. 12) in dieser Menge zu wissen. Zusatzlich werden andere Größen benötigt.

Eine der Größen neben den Kenngrößen der *tasks* ist die auf die *tasks* anzuwendende Planbarkeitsanalysemethode. Das Werkzeug kennt drei verschiedene Methoden: ratenmonotone Analyse (RMA, v. engl. R*ate* M*onotonic* A*nalysis*), Analyse für die Laufplanmethode ,,Planen nach Fristen" (EDFA, v. engl. E*arliest* D*eadline* F*irst* A*nalysis*) und Analyse der Antwortzeiten (RTA, v. engl. R*esponse* T*ime* A*nalysis*). Diese wurden bereits in Abschnitt 3 beschrieben.

Von der Laufplanmethode hängt ab, welche weiteren Kenngrößen für die Analyse benötigt werden. In den einfachen Formen von *RMA* und *EDF*, also ohne die Berücksichtigung von Betriebsmitteln, werden je *task* lediglich zwei Kenngrößen zur Ermittlung der Planbarkeit gebraucht. Es sind die Periode (T) und die maximale Ausführungszeit (C). Der Wert der Frist für *RMA* und *EDFA* ergibt sich aus der Periode (siehe Bedingung 3, Seite 17).

Die meisten Kenngrößen können für die Berechnung der Planbarkeit nach *RTA* anfallen. Hier sind es je *task* wenigstens vier Angaben: die Periode (T), die maximale Ausführungszeit (C), die Frist (D) und die Nummer(n) des oder der verwendeten Betriebsmittel (R). Die Anzahl der Daten einer *task* kann sich hier dann erhöhen, wenn diese *task* mehrere Betriebsmittel benutzen wird. Die maximale Ausführungszeit wird in diesem Fall unterteilt. Jeder Teil repräsentiert dann die Ausführungszeit in einem kritischen Abschnitt (der Zeitabschnitt, in dem ein Betriebsmittel gehalten wird), oder er repräsentiert die Ausführungszeit in einem unkritischen Abschnitt (Abschnitt außerhalb kritischer Abschnitte). Diese Teilausführungszeiten werden durch je eine Festpunktzahl repräsentiert. Die Anzahl dieser Teilausführungszeiten muß mit der Anzahl der kritischen und unkritischen Abschnitte übereinstimmen. Eine *task* wird immer mit der Ausführung eines unkritischen Abschitts beginnen. Nur in einem solchen kann sie ihr erstes Betriebsmittel anfordern. Es ist auch möglich, daß mehrere kritische Abschnitte ineinander verschachtelt sind. Hier kann dann ein neues Betriebsmittel innerhalb eines kritischen Abschnitts angefordert werden. Das wird in der Eingabe für den Analysator gesondert angegeben (siehe unten). Zu beachten ist, daß das zuletzt angeforderte Betriebsmittel als erstes wieder freigegeben werden muß. Die Algorithmen der Planbarkeitsanalyse sehen keine Überlappung vor. Abschnitt 2.6.1, S. 21 erklärt die Probleme bei der Betriebsmittelnutzung.

Die Einheit der Zeitangaben ist für die Berechnung belanglos. Sie dient nur dazu, bei den Ausgaben die Zeitwerte zu bezeichnen. Diese Einheit muß keine bekannte

Einheit sein. Hier ist jede erdenkliche Bezeichnung erlaubt, z. B. s, ms, Takte, Momente, 3_xtel. Die Beschränkung bei dieser Angabe ist die auf ein Literal.

Alle Angaben, die der Analysator für die Berechnung der Durchführbarkeit braucht, werden in einer Datei zusammengefaßt. Dabei stehen alle Kenngrößen, die zu einer *task* gehören, in einem Block. Die Reihenfolge der Kenngrößen in diesem Block ist beliebig. Die einzige Bedingung ist, daß die Angabe über die *task*-Nummer vor den anderen Größen steht, damit die Werte der Kenngrößen beim Einlesen zugeordnet werden können. Eine weitere, globalere Bedingung ist, daß alle *tasks* Angaben zu den gleichen Kenngrößen haben müssen, da die verwendeten Algorithmen das erwarten. Diese Dateien, in denen die Kenngrößen der *tasks* und die restlichen Angaben stehen, werden **Spezifikationsdateien** genannt. Ein typisches Beispiel für eine Spezifikationsdatei sieht so aus:

```
# data file for the anatool.

method: RTA
unit: ms
resource access protocol: inheritance

task:             1
period:           44
deadline:         44
computation time: 1.0, 2.0, 2.0
resource:         {1}, {2}, {0}

task:             2
period:           60
deadline:         60
computation time: 1.0, 9.0, 3.0, 3.0
resource:         {0}, {2}, {3}, {0}

task:             3
period:           80
deadline:         80
computation time: 8.0, 7.0,   4.0,  3.0, 3.5
resource:         {1}, {2}, {2, 4}, {2}, {0}
```

Beispiel einer Spezifikationsdatei für den Analysator.

Beschreibung der Spezifikationsdatei:

- `#`: Zeilen, die mit einem „#“ beginnen, sind Kommentarzeilen. Sie werden beim Einlesen ignoriert.
- Auch die Leerzeilen werden ignoriert.
- `method`: Hier sind zulässig: *RMA*, *RTA* und *EDFA*.
- `unit`: Die Angaben über die Einheit (engl. *unit*) wurden oben erläutert.
- `task`: Die Nummern der *tasks* müssen nicht geordnet sein. Es ist eine beliebige Reihenfolge zulässig. Auch müssen die *tasks* nicht von 1 an fortlaufend durchnumeriert sein. Es darf nur keine Nummer mehrfach vergeben werden.

- `period`: Die Periode ist ein Zahlenwert. An dieser Stelle kann ein Ganzzahl- oder ein Festpunktliteral stehen.

- `deadline`: Für die Frist (engl. *deadline*) gilt dasselbe wie für `period`.

- `computation time`: Die Ausführungszeit (engl. *computation time*) wird mit einem Wert angegeben, genau dann, wenn die entsprechende *task* kein Betriebsmittel haben möchte. Werden Betriebsmittel angefordert, dann gibt es für die entsprechenden kritischen und unkritischen Abschnitte zugehörige Teilausführungszeiten. Alle Teilausführungszeiten werden hier angegeben.

- `resource`: Die anzufordernden Betriebsmittel (Ressourcen) werden durch Ganzzahlen repräsentiert. Die Zahlen sind in geschweifte Klammern zu stellen. Eine „0“ bedeutet, daß an dieser Stelle kein Betriebsmittel gehalten wird, was als unkritischer Abschnitt bezeichnet wird. Geschachtelte kritische Abschnitte werden als Menge von Betriebsmittelnummern in ein geschweiftes Klammerpaar gestellt. Dabei bedeutet die Menge $\{2, 4\}$ in der Zeile `resource` im Block von *task* 3, daß innerhalb des kritischen Abschnitts mit Betriebsmittel 2 (während 2 gehalten wird) Betriebsmittel 4 angefordert wurde und nun gehalten wird. Diese Mengenangabe steht an dritter Stelle in der Zeile `resource`. Entsprechend an dritter Stelle in der Zeile der Ausführungszeiten (`computation time`) steht die zugehörige Ausführungszeit des kritischen Abschnitts mit Betriebsmittel 4 (innerhalb des kritischen Abschitts mit Betriebsmittel 2). Allgemein sind für alle Ausführungsabschnitte, ob kritisch oder unkritisch, die zugehörigen Ausführungszeiten anzugeben. Es müssen so viele Werte in der Zeile der Ausführungszeiten stehen wie es Mengen in der Zeile der Ressourcen gibt.

In Bild 8 wird gezeigt, wie die Angaben in der Spezifikationsdatei von oben zu deuten sind.

Die maximalen Ausführungszeiten der *tasks* ergeben sich aus der Summe der Teilausführungszeiten. *Task* 1 hat demnach eine maximale Ausführungszeit von $5,0$ Einheiten. Für jeden Abschnitt, ob kritischer oder unkritischer, wird die Ausführungszeit angegeben. *Task* 1 beginnt die Ausführung im kritischen Abschnitt von Betriebsmittel 1 (in der Datei angegeben durch $\{1\}$). Dort beträgt die Ausführungszeit $1,0$ Einheiten. *Task* 1 tritt dann unmittelbar in den kritischen Abschnitt von Betriebsmittel 2 ein und wird da für $2,0$ Einheiten ausgeführt. Nach diesem kritischen Abschnitt läuft sie noch $2,0$ Einheiten in einem unkritischen Abschnitt (bezeichnet durch $\{0\}$), bevor sie mit der gesamten Ausführung innerhalb der Periode von $44,0$ Einheiten fertig ist.

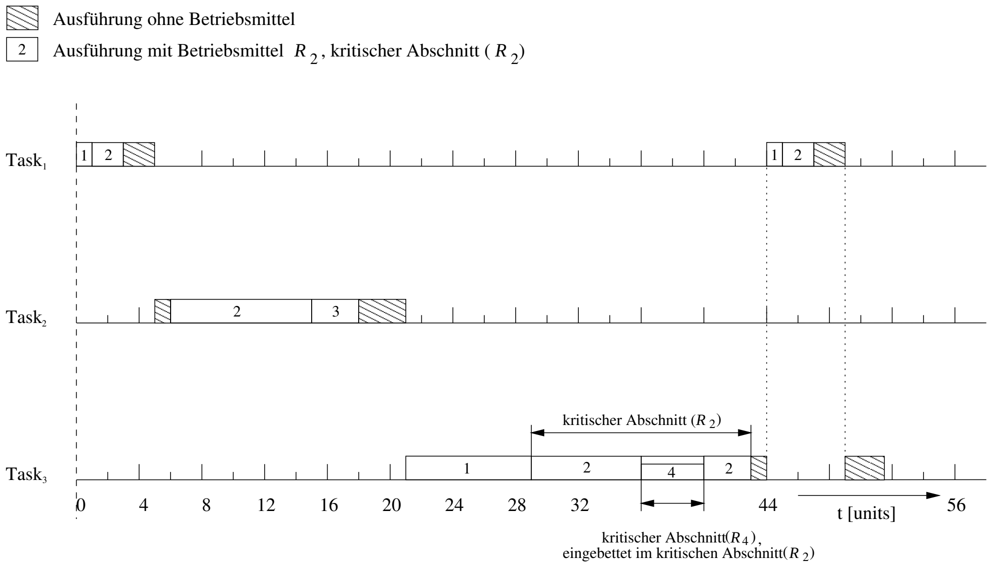

Bild 8: Bedeutung der Angaben in der Spezifikationsdatei

Task 3 wird Betriebsmittel 2 und 4 anfordern. Hier ist der geschachtelte Fall dargestellt. Während diese *task* sich im kritischen Abschnitt mit Betriebsmittel 2 befindet, tritt sie in einen kritischen Abschnitt mit Betriebsmittel 4.

Das Zugriffsprotokoll für Betriebsmittel ist wahlweise anzugeben. Fehlt die Zeile für das Protokoll (`resource access protocol`), so wird vom Analysator der voreingestellte Wert `inheritance` (Prioritätenerben), angenommen.

5 Ausgabe des Planbarkeitsanalysators

Umfang und Inhalt der Ausgabe des Planbarkeitsanalysators hängen ab von der verwendeten Analysemethode und von den Kenngrößen der zu untersuchenden *tasks*. Im wesentlichen gibt es aber nur zwei verschiedene Ergebnisse: ,,Die Menge der *tasks* ist planbar.“ (engl. `This set of tasks IS schedulable with` < *method* >.) und ,,Die Menge der *tasks* ist nicht planbar.“ (engl. `This set of tasks IS NOT schedulable with` < *method* >.) Es liegt an den in der Eingabe eingestellten Optionen, welche Erklärungen zusätzlich ausgegeben werden. Wird nichts eingestellt, kommen auch keine erklärenden Ausgaben. Ein Beispiel für eine Ausgabe mit Erklärungen zeigen folgende Zeilen:

```
Analysis method: RTA
```

```
Unit:  ms
Resource access protocol: inheritance

task number, now:  1; in input:  3
period:   40.000
deadline:   40.000
computation time:   1.000,    2.000,    2.000
sum of computation times: 5.000
resources:  {1}, {2}, {0}
priority:  4

task number, now:  2; in input:  2
period:   60.000
deadline:   60.000
computation time:    1.000,    9.000,    3.000,    3.000
sum of computation times: 16.000
resources: {0}, {2}, {3}, {0}
priority:  3

task number, now:  3; in input:  4
period:   80.000
deadline:   80.000
computation time:    8.000,    7.000,    4.000,    8.000
sum of computation times: 27.000
resources: {1}, {2}, {2, 4}, {2}
priority:  2

task number, now:  4; in input:  1
period:  100.000
deadline:  100.000
computation time:    4.000,    6.000,    5.000,    2.000,    3.000
sum of computation times: 20.000
resources: {0}, {1}, {2}, {4, 3}, {1}
priority:  1



R1[4] | R2[4] | R3[4] | R4[4] |
 ---------------------------------------
 1|      1.000   2.000   0.000   0.000
 2|      0.000   9.000   3.000   0.000
 3|      8.000  19.000   0.000   4.000
 4|      9.000   5.000   2.000   2.000
```

```
blocking factor task  1:   35.000
blocking factor task  2:   28.000
blocking factor task  3:    9.000
blocking factor task  4:    0.000

response time task  1:   40.000
response time task  2:   54.000
response time task  3:   78.000
response time task  4:  121.000  <-- greater than deadline of  100.000

This set of tasks IS NOT schedulable with RTA.
```

Die erste Zeile enthält die Angabe der zu verwendenden Planbarkeitsanalysemethode. Im Beispiel ist es *RTA* (`Analysis method: RTA`). Auch die Einheit wird immer angegeben, hier: ms (`Unit: ms`). Die darauffolgende Zeile gibt das anzuwendende Zugriffsprotokoll für die Betriebsmittelnutzung an. In diesem Falle ist es das „Protokoll des Prioritätenerbens“ (`resource access protocol: inheritance`).

Die Zeile „`task number, now: 1; in input: 3`“ zeigt zweierlei Numerierungen der *tasks* an: einmal die innerhalb der Spezifikation, zum anderen die innerhalb der Datenstruktur im Analyseprogramm. Die Numerierung innerhalb der Spezifikation legt der Erzeuger der Spezifikationsdatei fest. Außer bei *EDFA* werden die Datensätze dann nach dem Einlesen vom Analysator sortiert. Für *RMA* werden sie nach den Perioden und für *RTA* nach den (relativen) Fristen sortiert. Die *task* mit der kürzesten Periode (*RMA*) bzw. kürzesten Frist (*RTA*) erhält die kleinste Nummer.

Nach den Angaben über Periode (`period`), Frist (`deadline`), maximale Ausführungszeiten (`WCET`, v. engl. *Worst Case Execution Time*), Summe der maximalen Ausführungszeiten (maximale Gesamtausführungszeit) und Betriebsmittel (`resources`) für jede *task* folgt die jeder *task* zugeordnete Priorität. Die Priorität steht im Falle *RMA* im umgekehrten Verhältnis zur Periode, und bei *RTA* steht sie im umgekehrten Verhältnis zur relativen Frist. Bei *EDFA* können den *tasks* keine Prioritäten im voraus gegeben werden, da bei der *EDF*-Laufplanmethode die Prioritäten dynamisch, also während der Laufzeit zugewiesen werden. Für die Analyse ist in diesem Falle die Kenntnis über die Prioritäten nicht wichtig. Die drei Methoden wurden im Abschnitt 3 vorgestellt.

Die anschließende Tabelle dient der Berechnung der Blockierzeiten. Wenn mehrere *tasks* dasselbe Betriebsmittel anfordern, dann muß damit gerechnet werden, daß eine höherpriorisierte *task* τ_1 von einer niederpriorisierten *task* $\tau_2$blockiert wird. Das passiert dann, wenn τ_2 während der Benutzung eines Betriebsmittels von τ_1 suspendiert wurde und τ_1 dann auch auf das Betriebsmittel zugreifen möchte. τ_1 kann es nicht erhalten, da es schon von τ_2 gehalten wird. τ_2 setzt jetzt ihre Ausführung solange fort, bis sie das Betriebsmittel wieder abgibt. In dieser Zeit blockiert sie die Ausführung der höherpriorisierten *task* τ_1. Zur genaueren Erklärung dient Abschnitt 2.6.1 auf Seite 24. Desweiteren bieten die dokumentierten Testbeispiele ab Seite 58 weitere Erklärungen.

6 Entwurf des Planbarkeitsanalysators

Es folgt eine Beschreibung des Entwurfs des Planbarkeitsanalysators.

Der Grund, warum für die Ein- und Ausgaben sowie für die Variablen und Kommentare des Planbarkeitsanalysators die englische Sprache gewählt wurde, ist der, um mit den anderen Werkzeugen innerhalb des Projektes „ADAM“ einheitlich zu sein.

6.1 Aufgabe des Planbarkeitsanalysators

Die Aufgabe des Planbarkeitsanalysators ist es, die im letzten Abschnitt beschriebenen Planbarkeitsanalyseverfahren automatisch durchzuführen. Die groben Anforderungen an das Werkzeug sind:

1. Einlesen der Kenngrößen der zu untersuchenden *tasks*,
2. Analyse der Kenngrößen mit den oben beschriebenen Methoden,
3. Aussage über die Durchführbarkeit.

Da das Werkzeug im Rahmen eines größeren Projektes entwickelt wurde (und weiterentwickelt wird), ist es nur ein Teil einer umfangreicheren Analyseumgebung. Die einzelnen Komponenten und ihre Anordnung sind in Bild 9 auf Seite 47 zusammengestellt.

Wie im Bild 9 zu sehen ist, stellt das Planbarkeitsanalysewerkzeug die letzte Stufe im gesamten Analysesystem dar. Es ist somit auf die Berechnungen der davorliegenden Stufen angewiesen. Es ist aber möglich –für Probezwecke– das Werkzeug

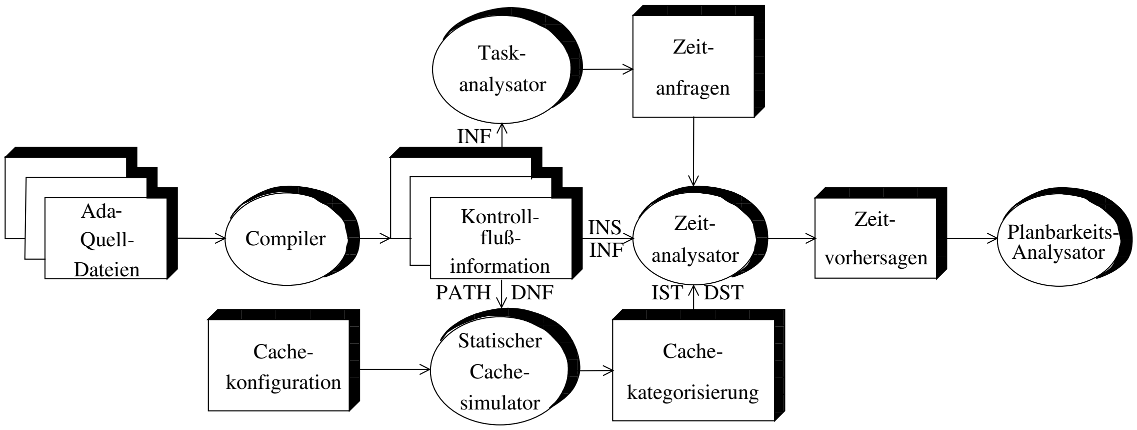

Bild 9: Analyseumgebung

unabhängig von den anderen Programmen zu starten. Als Eingabe dienen dann fiktive Werte. Die Ausgabe der anderen Stufen sind die maximalen Ausführungszeiten der zu untersuchenden *tasks*. Diese Zeiten, zusammen mit den restlichen Kenngrößen der *tasks* wie Periode, Frist und anzufordernde Betriebsmittel sowie *task*-übergreifende Angaben wie Laufplanmethode, Einheit der Zeitangaben und Zugriffsprotokoll für die Betriebsmittel, bilden die Eingabe des Planbarkeitsanalysators.

6.2 Eingabe des Planbarkeitsanalysators

Eine Struktur der Eingabewerte war nicht vorgegeben. Bis zum jetzigen Zeitpunkt ist es auch nicht möglich, Daten von den anderen Stufen (Analysatoren) zu erhalten, da sich diese erst in der Entwicklung befinden und noch keine Ausgaben generieren. Dadurch konnte beim Entwerfen des Planbarkeitsanalysators eine beliebige Eingabentruktur angenommen werden. Später wurde die nun gültige Form herausgearbeitet und stellt jetzt die Eingabeschnittstelle des Planbarkeitsanalysators dar. Abschnitt 4 beschreibt im Detail die Eingabeschnittstelle.

Zunächst gab es die Idee, die Syntax der Eingabeschnittstelle mit Hilfe einer sogenannten „*Backus-Naur-Form*“ (BNF) zu beschreiben. Eine BNF wird oft dann eingesetzt, wenn Schnittstellen eindeutig zu beschreiben sind. Mit ihr kann man aus Zeichen (Buchstaben und Zahlen) zu bildende Strukuren genau definieren. Das ist jedoch nicht für beliebige Strukturen möglich. Wenn bestimmte Zeichen in einer Strukur in besonderer Beziehung zu bestimmten anderen Zeichen stehen sollen (d. h. daß diese Zeichen in einem bestimmten **Kontext** stehen müssen), dann ist es schwierig, diese Struktur mit der „*Backus-Naur-Form*“ zu beschreiben. Es wäre zunächst eine BNF ohne Beachtung des Kontexts zu erstellen. Zusätzlich und extra müßte dann der Kontext beschrieben werden.

Weil die Eingaben für den Planbarkeitsanalysator in der Spezifikationsdatei in einem Kontext stehen und dadurch die Beschreibung durch BNF unnötig kompliziert wäre, wurde auf die Beschreibung durch BNF verzichtet. Im Abschnitt 4 wurde die Eingabeschnittstelle mit Worten und am Beispiel erklärt. Diese Erklärung sollte ausreichen, um die Korrektheit einer Spezifikationsdatei als Eingabeschnittstelle zu erkennen.

Der Planbarkeitsanalysator überprüft nicht zuerst die Richtigkeit der Syntax der Eingabe. Wenn ein Schlüsselwort (z. B. Bezeichner für eine Kenngröße einer *task*, etwa `period:`) eingelesen wurde, wird anschließend versucht, einen zugehörigen Wert einzulesen. Folgt dem Schlüsselwort etwas anderes als der Wert eines erwarteten Typs, dann bricht das Programm mit einer Fehlermeldung ab. Die Eingaben werden also unvorbereitet eingelesen.

Im Normalfall sollten keine fehlerhaften Eingaben zu erwarten sein, da die Spezifikationsdateien von den anderen Werkzeugen, die dem Planbarkeitsanalysator vorangestellt sind (s. S. 47), erzeugt werden.

6.3 Ausgabe des Planbarkeitsanalysators

Die Ausgaben des Planbarkeitsanalysators haben keine feste Struktur. Die Struktur wird von der Eingabe stark beeinflußt. So kann eine Ausgabe aus nur einer Zeile bestehen oder aus vielen Seiten dokumentierter Ergebnisse. Auch hierfür gibt es keine Beschreibung in „*Backus-Naur-Form*“. Abschnitt 5 beinhaltet die Beschreibung der Ausgaben mit Beispiel.

6.4 Programmiersprache

Die Aufgabe, ein Werkzeug zur Analyse der Planbarkeit von Ada-*tasks* zu erstellen, legt nahe, sich bei der Wahl der Programmiersprache, in der das Werkzeug geschrieben werden soll, für Ada zu entscheiden. Es war nicht zu erwarten, daß das Werkzeug besondere Anforderungen an die zu verwendende Programmiersprache stellen wird. Deshalb war es grundsätzlich möglich, eine beliebige Sprache zu benutzen. Da der Verfasser dieser Arbeit Erfahrungen mit Ada hatte, entschied er sich auf Anhieb dafür.

7 Umsetzung des Entwurfs

Alle Programmteile stehen zusammen in einer Datei. Es war nicht einzusehen, die verhältnismäßig einfache Struktur des Quellprogramms in verschiedene Dateien aufzuteilen. Die Handhabbarkeit der Datei wird auch durch die Größe von rd. 38kB nicht beeinträchtigt. Für selbstdefinierte Pakete gab es keine Verwendung. Obwohl mit dem Analysator *tasks* untersucht werden, die in Ada geschrieben sein und nebenläufig in Echtzeitsystemen laufen können, gibt es im Analysator selbst keine selbsterzeugten *tasks*. Auch geschützte Objekte findet man hier nicht.

7.1 Wichtige Variablen

In diesem Abschnitt sollen die Hauptvariablen beschrieben werden. Es gibt zwei globale Variablen. Die erste beinhaltet alle Informationen über die zu untersuchenden *tasks*. Sie heißt `Task_Data()` und ist vom selbstdefinierten Typ `Task_Rec`. Zunächst werden die Kenngrößen der *tasks* (*task*-Nummer, Periode, Frist, Ausführungszeit, Betriebsmittel) dort hereingeladen. Später kommen die Größen, die im Programm errechnet werden (Priorität, Summe der Teilausführungszeiten bei Betriebsmittelnutzung, Startzeit sowie Hilfswerte für die Berechnungen), hinzu. Die zweite globale Variable heißt `N`. Sie ist eine Variable vom Typ `Integer` und enthält die Anzahl der *tasks*, die untersucht werden sollen.

Diese beiden Variablen wurden global vereinbart, um sie nicht bei jedem Prozedur- oder Funktionsaufruf zu übergeben. Sie werden in fast jeder Unterprozedur oder Unterfunktion benutzt. Sie müssen also fast überall bekannt sein. Der Umgang mit ihnen wird dadurch, daß sie global sind, etwas handhabbarer.

Eine andere Variable, die fast überall bekannt sein muß, ist die Variable `Common_Spec`. Der Name soll darauf hinweisen, daß es sich um eine gemeinsam (engl. *common*) benutzte Variable handelt, die Spezifikationen (engl. *specification*) enthält. In dieser Variablen sind enthalten: Methode der Planungsanalyse, Einheit der Zeitwerte, Anzahl der insgesamt verwendeten Betriebsmittel, das anzuwendende Zugriffsprotokoll für die Betriebsmittel sowie einige Hilfswerte. Die Zeichenkette des Ergebnisses (`This set of tasks IS (NOT) schedulable with`, der Rest wird bei der Ausgabe angefügt) wird auch hier gespeichert.

Die Variable `Switch` beinhaltet den Laufplan für untersuchte *tasks* nach der ratenmonotonen Methode bis zum Ende der kritischen Zeitzone. `Switch` wird nur dann verwendet, wenn die *RMA*-Methode eingestellt ist und die *tasks* keine Betriebsmittel nutzen. Der Ablauf der *tasks*, einschließlich etwaiger Fristüberschreitungen, kann vom Analysator ausgegeben werden. `Switch` ist vom selbstdefinierten Typ `Switch_Type`. Dieser ist ein Reihungstyp mit Komponenten vom Typ `Switch_Rec`.

Erklärung des Typs `Switch_Rec`:

```
type Switch_Rec is record
   Whenn       : Real;       --> Zu welchem Zeitpunkt ...
   Which       : Natural;    --> ... macht welche task ...
   What        : Events;     --> ... was (Ready, Started, Continued,
                                      Finished, Suspended, Missed).
end record;
```

Dabei bedeuten:

`Ready`: *task* wurde bereit,

`Started`: *task* wurde gestartet,

`Continued`: Ausführung der *task* wird fortgesetzt,

`Finished`: Ausführung der *task* ist beendet,

`Suspended`: *task* wurde suspendiert,

`Missed`: *task* hat eine Frist verpaßt.

Wann ein Ereignis auftrat wird in der Komponenten `.Whenn` gespeichert, und welche *task* davon betroffen war, wird in `.Which` geschrieben.

7.2 Einlesen der Eingabe

Für das Einlesen der Spezifikation wurden individuelle Einleseroutinen geschrieben, die sich leicht erweitern lassen. Das Einlesen geschieht in zwei Stufen. In der ersten Stufe wird ein Schlüsselwort und in der zweiten Stufe wird der zugehörige Wert eingelesen. Die Schlüsselwörter sind die Bezeichnungen der Kenngrößen der *tasks*. Es gibt insgesamt 5 Kenngrößen:

- die Nummer der *task*, in der Eingabe bezeichnet mit `task:`,
- die Periode der *task*, bezeichnet mit `period:`,
- die Frist der *task*, bezeichnet mit `deadline:`,
- die Ausführungszeit(en) der *task*, bezeichnet mit `computation time:`,
- die von der *task* anzufordernden Betriebsmittel (Ressourcen), bezeichnet mit `resource:`

Zusätzlich sind in der Eingabedatei enthalten:

- die anzuwendende Analysemethode, bezeichnet mit `method:`,
- die Einheit der Zeitangaben in Eingabe und Ausgabe, bezeichnet mit `unit:`,
- das Betriebsmittelzugriffsprotokoll, bezeichnet mit `resource access protocol:`

Die Struktur der Eingabe ist in Abschnitt 4 beschrieben. Eine Spezifikationsdatei, die die Eingabe des Analysators enthält, wird von oben nach unten eingelesen. Hierzu dient die Unterprozedur `Load_Task_Data`. In dieser Unterprozedur werden alle Kenngrößen der *tasks* in die Komponenten der globalen Variablen `Task_Data`, und die zusätzlichen Größen werden in die „Sammelvariable“ `Common_Spec` eingelesen.

Wie oben bereits erwähnt, gibt es zwei Stufen, in denen eine Spezifikationsdatei eingelesen wird. In der ersten Stufe werden solange Zeichen aus der Eingabedatei gelesen, bis ein Schlüsselwort erkannt wurde.

Diese erste Stufe wurde in der Funktion `scan_Parameter_Name` realisiert. Es ist eine Unterfunktion innerhalb der Unterprozedur `Load_Task_Data`. Die Schlüsselwörter werden hier als Parameternamen bezeichnet. Die Namen aller möglichen Schlüsselwörter (Parameternamen) sind in einer Konstanten (`Param_Name`) gespeichert. In der Spezifikationsdatei (siehe Abschnitte 4 und 6.2) folgt jedem Schlüsselwort ein Doppelpunkt („:“). Das ist die Endemarkierung für die Schlüsselwörter. Es werden alle Zeichen vom Zeilenanfang bis zum Doppelpunkt gelesen. Anschließend wird die so eingelesene Zeichenkette mit den Zeichenketten in der Konstanten `Param_Name` verglichen. Kommt es zu keiner Übereinstimmung, wird eine Fehlermeldung ausgegeben, und das Programm bricht ab. Wird die eingelesene Zeichenkette erkannt, wird aus einer anderen Konstanten eine zugehörige Kennmarke als Rückgabewert an die aufrufende übergeordnete Unterprozedur `Load_Task_Data` zurückgegeben. Die Verwendung solcher Kennmarken ist notwendig, da die später auszuführende `case`-Anweisung diese Kennmarken zur Entscheidung für einen weiteren Schritt braucht.

Die zweite Stufe des Einlesens der Spezifikationsdatei versucht, die Werte der Kenngrößen einzulesen. Das geschieht in einer `case`-Anweisung. Sie steht in der Unterprozedur `Load_Task_Data`. Entsprechend der Kennmarke werden bestimmte Datentypen erwartet und einzulesen versucht.

Beispiel:

```
method:  RMA
unit:    ms

task:    1
period:  20

      ⋮
```

Beispiel eines gültigen Teils einer Spezifikationsdatei für den Analysator.

Stufe 1 liest die Zeichen des ersten Wortes `method` der Spezifikationsdatei bis zum Doppelpunkt ein. Die Zeichenkette wird mit den in der Konstanten `Param_Name` vorhandenen Zeichenketten verglichen. Dort existiert die Zeichenkette `method`. Eine Übereinstimmung wird also festgestellt. Aus der Konstanten `Parameter` vom Aufzählungstyp `Params` wird das entsprechende Literal (`method`) als Rückgabewert der Funktion `Scan_Parameter_Name` an die aufrufende, übergeordnete Prozedur `Load_Task_Data` geliefert. Die aufrufende Prozedur ist Stufe 2 des Einlesens. Hier wird in der `case`-Anweisung die Kennmarke `method` erkannt (`when method => ...`) Die Anweisungen, die auf `when method =>` folgen, versuchen, eines der Literale `RMA`, `EDFA` und `RTA` (vom Aufzählungstyp `Method_Type`) einzulesen. Das gelingt im Beispiel. Anschließend wird wieder die Funktion `Scan_Parameter_Name` aufgerufen. Dort wird das nächste Wort, `unit`, eingelesen. Zurückgegeben wird die Kennmarke `unit`, und die `case`-Anweisung in `Load_Task_Data` verzweigt, entsprechend der Kennmarke `unit`, zu den Anweisungen zum Einlesen eines Literals. Wieder wird `Scan_Parameter_Names` aufgerufen. Die Leerzeile wird übersprungen, `task` eingelesen, die folgende `1` auch und so weiter.

Alle eingelesenen Kenngrößen werden der *task* mit der zuletzt eingelesenen Nummer zugeordnet. Die Werte der Kenngrößen werden zusammen mit der Nummer der *task* in dieselbe Komponente der Reihung `Task_Data` geschrieben.

7.3 Analyse

Nach dem Laden der Spezifikationsdaten wird dann in der Hauptprozedur entsprechend der Analysemethode weiterverfahren. Es wird durch eine `case`-Anweisung in einen bestimmten Ausführungsteil verzweigt.

RMA

Vom *RMA*-Zweig aus werden die *tasks* zunächst sortiert. Dazu wird in der Unterprozedur `Sort_Task_Data` die Reihung `Task_Data` mit den *task*-Kenngrößen derart sortiert, daß die erste Komponente die *task* beinhaltet, die die kleinste Periode hat. Folgende Komponenten beinhalten *tasks* mit größeren Perioden. Als Sortieralgorithmus dient der einfache *bubble-sort*-Algorithmus.

Anschließend werden durch Aufruf der Unterprozedur `Set_Prio` den *tasks* mit den kürzeren Perioden höhere Prioritäten zugewiesen (siehe Abschnitt 3.4). Die Nummern der Prioritäten werden in den Komponenten der Variablen `Task_Data`, also zu den entsprechenden *tasks* gespeichert.

Wenn keine *task* Betriebsmittel anfordert, wird die Standard-*RM*-Analyse (siehe Abschnitt 3.4) durchgeführt. Eine Bedingung der Standard-*RM*-Analyse für die Planbarkeit einer Menge von *tasks* ist, daß die *tasks* der Menge innerhalb der kritischen Zeitzone (siehe Abschnitt 3.2) planbar sein müssen.

Das Errechnen der kritischen Zeitzone erledigt die Unterprozedur `Compute_Crit_Time_Zone`. Diese beinhaltet einige Unterprozeduren und -funktionen. Die Berechnung ist umfangreich. Sie ist eine Simulation des ratenmonotonen Laufplaners. Für jede *task*, oder genauer: für jede Instanz einer *task*, werden die Start-, Suspendier-, Beendigungs- und Laufzeiten ausgerechnet. Zu den Start-, Suspendier- und Beendigungszeiten erfolgen Kontextwechsel (siehe Seite 19). Diese werden in einer Reihungsvariablen namens `Switch`, die Komponenten eines Verbundtyps hat, gespeichert und stellen den Ablauf der Laufplanung bis zum Ende der kritischen Zeitzone dar.

Folgende Skizze gibt einen Überblick über den Algorithmus zum Berechnen der kritischen Zeitzone:

```
begin Compute_Crit_Time_Zone
   loop
```

```
      Switch[Aktuelle_Position].Whenn := Aktuelle_Zeit;
      Switch[Aktuelle_Position].Which := Aktuelle_Task;

      if Suspendiert(Aktuelle_Task) and
         Verbleibende_Ausfuehrungszeit(Aktuelle_Task) <
            Gesamtausfuehrungszeit(Aktuelle_Task)
      then
         Switch[Aktuelle_Position].What := Fortgesetzt;
      else
         Switch[Aktuelle_Position].What := Gestartet;
      end if;

      Aktuelle_Position := Aktuelle_Position + 1;
      Ermittle_Naechste_Task;
      if Hoeherprio_Task(Start) < Aktuelle_Task(Ende)
      then
         Vergangene_Zeit := Hoeherprio_Task(Start) - Aktuelle_Zeit;
         Verbleibende_Ausfuehrungszeit(Aktuelle_Task) :=
            Verbleibende_Ausfuehrungszeit(Aktuelle_Task) -
            Vergangene_Zeit;
         Suspendiert(Aktuelle_Task);
         Aktuelle_Zeit := Aktuelle_Zeit + Vergangene_Zeit;
         Naechste_Task := Hoeherprio_Task;
      else
         Aktuelle_Zeit := Aktuelle_Zeit +
            Verbleibende_Ausfuehrungszeit(Aktuelle_Task);
         Verbleibende_Ausfuehrungszeit(Aktuelle_Task) :=
            Gesamtausfuehrungszeit(Aktuelle_Task);
         Nicht_Suspendiert(Aktuelle_Task);
         Anzahl_Der_Ausfuehrungen(Aktuelle_Task) :=
            Anzahl_Der_Ausfuehrungen(Aktuelle_Task) + 1;
         Naechster_Start(Aktuelle_Task) :=
            Anzahl_Der_Ausfuehrungen(Aktuelle_Task) *
            Periode(Aktuelle_Task);
         Switch.What := Beendet;
      end if;

      Aktualisiere_Startzeiten_Aller_Tasks;
      Ueberpruefe_Verpasste_Fristen(Alle_Tasks);

   exit_from_loop when Abgearbeitet(Letzte_Task);
      Aktuelle_Task := Naechste_Task;
   end loop;
```

```
33    if Auszugeben
34    then
35       Gib_Aus(kritische_Zeitzone);
36    end if;

37 end Compute_Crit_Time_Zone;
```

Algorithmus der Berechnung kritischer Zeitzonen

Zunächst werden die aktuelle Zeit und die Nummer der aktuellen *task* gespeichert (Zeilen 3 und 4). Im ersten Durchlauf der Schleife sind das die Startzeit (`0.0`) und die Nummer der *task* mit der höchsten Priorität (oft `1`). In Zeile `5` wird geprüft, ob die aktuelle *task* bereits unterbrochen wurde und nun wieder zur Ausführung kommen soll. Ist das der Fall, wird das vermerkt (Z. 7). Ist die aktuelle *task* nicht suspendiert (unterbrochen), dann wird sie neu gestartet (wiederholt oder zum ersten Mal). Auch das wird vermerkt (Z. 9). In Zeile 14 wird entschieden, ob die aktuelle *task* unterbrochen wird. Sie wird unterbrochen, wenn eine höherpriorisierte *task* aktiv wird, bevor die aktuelle beendet ist. Zeilen 15–19 behandeln den Fall, in dem die aktuelle *task* von einer anderen *task* unterbrochen wird, und in den Zeilen 21–26 wird der andere Fall behandelt, in dem die *task* bis zum Ende läuft.

Die kritische Zeitzone ist vollständig berechnet, wenn die *task* mit der längsten Periode und der geringsten Priorität bis zum Ende der Ausführungzeit durchgerechnet wurde. Es kann dann schon entschieden sein, ob die untersuchte Menge der *tasks* planbar ist oder nicht. Wenn bei der Berechnung der kritischen Zeitzone, was eine Simulation der Laufplanung darstellt, eine *task* ihre Frist verpaßt hat, ist die Menge nicht planbar.

Nach der anschließenden Berechnung der Prozessorbelastung (siehe dazu S. 26) wird die Formel der ratenmonotonen Analyse (siehe S. 34) angewandt. Damit ist die ratenmonotone Analyse für den Fall ohne Benutzung von Betriebsmitteln abgeschlossen. Hierauf folgt die Ausgabe des Ergebnisses: Die Menge der *tasks* ist (nicht) planbar.

Im Falle der Nutzung von Betriebsmitteln wird nicht die kritische Zeitzone ermittelt, da sie für die *RM-Analyse mit Blockierzeiten* belanglos ist. Außerdem wird ein anderer *UB-Test* durchgeführt (siehe Abschnitt 3.4). Hier werden auch die Prozessorbelastungen der einzelnen *tasks* anders berechnet, wobei zusätzlich die Blockierzeiten berücksichtigt werden.

Je nach Einstellung des Ausgabemodus (Zeile `info output mode:` in der Spezifikationsdatei) werden die kritische Zeitzone, die Prozessorbelastung(en) und die Kenngrößen der *tasks* ausgegeben.

EDFA

Für die Laufplanmethode ,,Planen nach Fristen" wurde bisher nur eine einfache Analyse (siehe Abschnitt 3.5) implementiert. Da sie die Planbarkeitsentscheidungen während der Laufzeit fällt, also *online* arbeitet, ist es schwierig, bei der *offline*-Analyse Zusatzgrößen wie Blockierzeiten zu berücksichtigen. Es wird angestrebt, die Analyse in der Zukunft auszubauen.

Zunächst wird die Prozessorbelastung berechnet. Danach wird überprüft, ob der Wert der Belastung kleiner oder gleich 1,0 (= 100% Prozessorbelastung) ist. Damit ist die Analyse abgeschlossen. Auch hier bestimmt der Ausgabemodus den Umfang der Ausgaben als Informationen über die Analyse.

RTA

Zuerst werden, wie bei *RMA* auch, die *tasks* sortiert. Im Unterschied zu *RMA* bekommen bei *RTA* aber dann die *tasks* mit den früheren Fristen höhere Prioritäten. Daß hier nach den Fristen sortiert werden soll, wird der Unterprozedur für das Sortieren (`Sort_Task_Data`) durch den Parameter `Common_Spec.Method` mitgeteilt. Dieser enthält den Namen der zu untersuchenden Laufplanmethode. Die Unterprozedur für das Zuweisen der Prioritäten (`Set_Prio`) gibt dann einfach den sortierten *tasks* der Reihe nach absteigende Prioritäten.

Sollen eine oder mehrere *tasks* Betriebsmittel benutzen, dann werden die Blockierzeiten berechnet. Anschließend wird die Unterprozedur `Rt_Analysis` aufgerufen, in der die komplexe Planbarkeitsanalyse nach der *RTA*-Methode durchgeführt wird.

Die Unterprozedur `Rt_Analysis` beinhaltet den Algorithmus zur Berechnung der Antwortzeiten der *tasks*, sowie zur Überprüfung der Antwortzeiten, ob sie innerhalb der Fristen der *tasks* liegen. Dieser Algorithmus (siehe S. 38) berechnet die Antwortzeiten iterativ, d. h. die rechte Seite der Gleichung wird wiederholt in die linke Seite eingesetzt, bis sich der Wert beider Seiten gleicht oder der Wert der rechten Seite größer als der Wert der Frist der untersuchten *task* ist.

Ist die Antwortzeit einer *task* zu lang, so daß die Frist der *task* verpaßt würde, bricht der Algorithmus ab und gibt entsprechend der Einstellung in der Zeile `info output mode` in der Spezifikationsdatei die Nummer und die Antwortzeit der *task* aus, die ihre Frist verpaßt hätte (wenn sie tatsächlich ausgeführt worden wäre).

In der `case`-Anweisung in der Hauptprozedur folgt nach dem `when`-Zweig für die *RT*-Analyse noch der Alternativzweig (`others => ...`) Hierhin wird verzweigt, wenn nicht eines der drei gültigen Analysemethoden, *RMA*, *EDFA* und *RTA*, angegeben wurde. Die Behandlung besteht in der Ausgabe einer Fehlermeldung, in der darauf hingewiesen wird, daß es sich bei der Angabe der Analysemethode um eine ungültige handelt.

Ausnahmebehandlung

Alle Ausnahmebehandlungen sind am Schluß der Hauptprozedur zusammengefaßt. Drei selbstdefinierte Ausnahmen werden behandelt:

- *Input_Error*: Diese Ausnahme wird ausgelöst, wenn die Angaben für die Ausführungszeit oder die für die Betriebsmittel nicht durch Kommas getrennt sind (z. B. wenn eines vergessen wurde) oder die Klammern bei der Angabe der Betriebsmittel falsch gesetzt sind. Solche Fehler können in handerstellten Eingabedateien enthalten sein. Eine Position des Fehlers in der Eingabedatei wird nicht ausgegeben.
- *Deadline_Assignment_Error*: Ist in der Spezifikationsdatei (Eingabedatei) die Analysemethode *RTA* eingestellt, aber sind nicht allen *tasks* Fristen zugewiesen worden, erfolgt diese Ausnahme mit Hinweis auf eine ordentliche Zuweisung von Fristen (Fristenzuweisung = engl. *deadline assignment*).
- *Syntax_Error*: Wenn in der Spezifikationsdatei ein ungültiges Schlüsselwort auftaucht, wird diese Ausnahme ausgelöst. Dabei wird das ungültige Wort mit ausgegeben.

Alle anderen Ausnahmen werden noch nicht gesondert behandelt. Später wird auch darauf eingegangen werden.

8 Benutzung des Planbarkeitsanalysators

Die Anwendung des Planbarkeitsanalysators ist einfach. Der Analysator arbeitet nicht interaktiv, und es gibt keine Kommandozeilenoptionen. Die Eingaben bestehen aus den Kenngrößen der zu untersuchenden *tasks* und den Angaben über die Verfahrensweise bei der Analyse (siehe Abschnitte 4 und 6.2). Die Eingaben werden über die Standardeingabe an den Analysator geleitet. Dadurch ist es leicht möglich, ihn in einer Kette zusammen mit den anderen Analysewerkzeugen (*pipeline-*, *cache-* und Zeitanalysator, siehe S. 47) einzusetzten. So können die Ausgaben einer Stufe als Eingabe an die nächste Stufe über die Standardein- und -ausgabe geschickt werden, was sehr leicht zu realisieren ist.

Wenn keine Eingabedatei vorliegt, kann eine eigene mittels eines Texteditors erstellt werden. Möglicherweise können Kenngrößen von bereits geplanten *tasks* gefunden oder sich neue ausgedacht werden. Wie auch immer, die Kenngrößen der *tasks* müssen dann, wie in Abschnitt 4 aufgezeigt, angeordnet werden. Ist eine Datei SPEZDATEI vorhanden, in der die benötigten Größen enthalten sind, kann der Analysator diese auswerten mit dem Kommando:

```
# anatool < SPEZDATEI
```

Die Ausgabe liefert das Ergebnis der Planbarkeitsanalyse oder, im Falle einer fehlerhaften Datei SPEZDATEI, eine Fehlermeldung. Der Umfang der Ausgabe hängt von der Eingabe ab (siehe Abschnitte 5 und 6.3). Mit der Ausgabe ist die Ausführung des Analysators abgeschlossen.

Im folgenden wird der Umgang mit mit dem Analysator an einigen Beispielen gezeigt.

8.1 Testbeispiele

Gegeben seien 3 *tasks* mit folgenden Kenngrößen:

τ_1: $T_1 = 4,0$; $C_1 = 1,5$

τ_2: $T_2 = 7,0$; $C_2 = 2,0$

τ_3: $T_3 = 12,0$; $C_3 = 4,0$

Ziel soll sein, die Planbarkeit dieser 3 *tasks* durch Anwendung der *RM*-Analyse zu testen. Es wird die kritische Zeitzone berechnet werden, und diese soll ausgegeben werden. Als Einheit für die Zeitangaben sei ms (Millisekunden) vorgegeben. Die entsprechende Spezifikationsdatei sieht so aus:

```
# data file for the anatool.
```

```
method: RMA
unit: ms
info output mode: zone

task:             1
period:           4
computation time: 1.5

task:             2
period:           7
computation time: 2

task:             3
period:           12
computation time: 4
```

Beispieldatei `rma_test1.dat`

Der Name der Datei sei `rma_test1.dat`. Die Analyse wird gestartet mit:

```
# anatool < rma_test1.dat.
```

Die Ausgabe dazu sieht dann so aus:

```
at    0.000 task 1  STARTED
at    1.500 task 1  FINISHED
at    1.500 task 2  STARTED
at    3.500 task 2  FINISHED
at    3.500 task 3  STARTED
at    4.000 task 3  SUSPENDED
at    4.000 task 1  STARTED
at    5.500 task 1  FINISHED
at    5.500 task 3  CONTINUED
at    7.000 task 3  SUSPENDED
at    7.000 task 2  STARTED
at    8.000 task 2  SUSPENDED
at    8.000 task 1  STARTED
at    9.500 task 1  FINISHED
at    9.500 task 2  CONTINUED
```

```
at   10.500 task 2  FINISHED
at   10.500 task 3  CONTINUED
at   12.000 task 3  SUSPENDED
at   12.000 task 1  STARTED  <- Task 3: deadline of  12.000 missed
at   13.500 task 1  FINISHED
at   13.500 task 3  CONTINUED
at   14.000 task 3  FINISHED

Critical time zone ranges from 0.000 to 14.000.

All time data in ms.

This set of tasks IS NOT schedulable with RMA.
```

Außer dem Ergebnis, das durch den Satz `This set of tasks IS NOT schedulable with RMA.` (dt. ,,Diese *task*-Menge ist nicht mit RMA planbar.") wiedergegeben wird, wird die kritische Zeitzone ausgegeben. Das war durch `info output mode: zone` (dt. ,,Info-Ausgabemodus: [kritische Zeit-]zone) in der Eingabe angegeben. Um den Sinn der kritischen Zeitzone zu verdeutlichen, soll Bild 10 genügen.

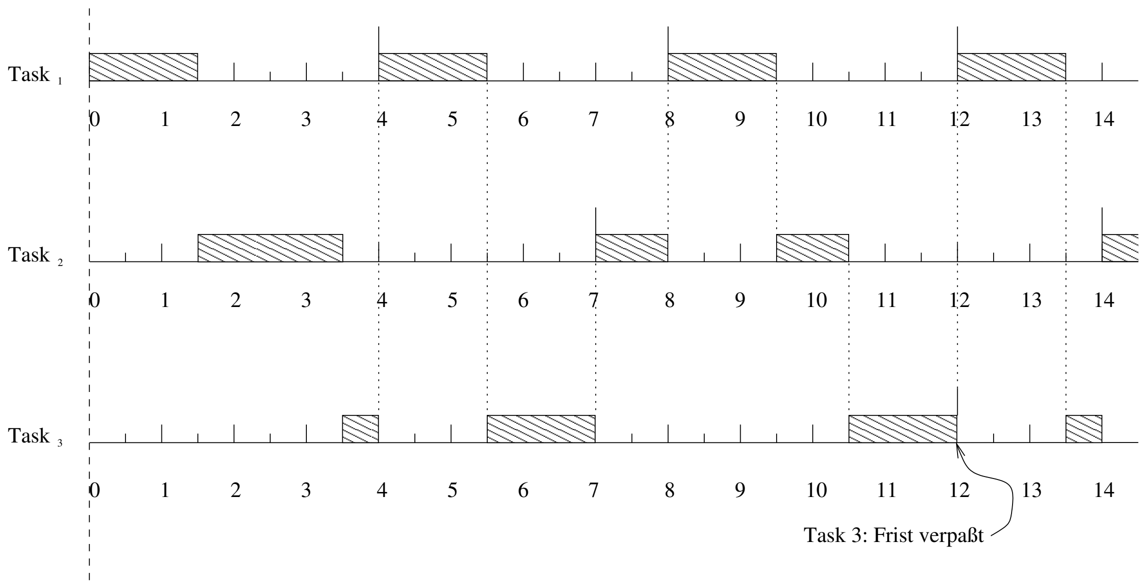

Bild 10: kritische Zeitzone von `rma1_test.dat`

Die Menge der *tasks*, die in der Spezifikationsdatei beschrieben wurde, ist also nicht planbar. Durch die Ausgabe der kritischen Zeitzone kann man sofort die *task* finden, die dafür verantwortlich ist.

Im zweiten Beispiel wird keine Frist verpaßt, aber es wird die Prozessorbelastungsgrenze überschritten (siehe Abschnitt 3.1):

```
# data file for the anatool.
info output mode: full

method: RMA
unit: ms

task:             1
period:           3
computation time: 1

task:             2
period:           4
computation time: 2
```

Beispieldatei `rma_test2.dat`

Nach

```
# anatool < rma_test2.dat
```

erhält man

```
Analysis method: RMA
Unit: ms

task number, now:  1; in input:  1
period: 3.000
computation times: 1.000
sum of computation times: 1.000
priority:  2

task number, now:  2; in input:  2
period: 4.000
computation times: 2.000
sum of computation times: 2.000
priority:  1

at    0.000 task 1  STARTED
```

```
at    1.000 task 1  FINISHED
at    1.000 task 2  STARTED
at    3.000 task 2  FINISHED

Critical time zone ranges from 0.000 to 3.000.

Utilization of  0.833 exceeds the utilization bound of 0.828.

All time data in ms.

This set of tasks IS NOT schedulable with RMA.
```

Hier werden innerhalb der kritischen Zeitzone keine Fristen verpaßt, also kann man davon ausgehen, daß auch in anderen Zeitbereichen keine *task* ihre Frist verpaßt (siehe dazu Abschnitt 3.4). Durch zu knappe Perioden bzw. zu langen Ausführungszeiten wird der Wert der Prozessorbelastung über den Wert der Prozessorbelastungsgrenze steigen, so daß keine sichere Laufplanung mehr garantiert werden kann. Das Ergebnis lautet daher: `This set of tasks IS NOT schedulable with RMA.` Die Prozessorbelastung errechnet sich nach der Formel

$$\sum_{i=1}^{2} \frac{C_i}{T_i}$$

wie folgt:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{3} + \frac{2}{4} = \frac{2}{6} + \frac{3}{6} = \frac{5}{6} \approx \underline{\underline{0,833}}.$$

Die Prozessorbelastungsgrenze von rd. $0,828$ wird durch

$$2\,(2^{\frac{1}{2}} - 1) = 2\,(\sqrt{2} - 1) \approx 2\,(1,414 - 1) \approx 2 \cdot 0,414 \approx \underline{\underline{0,828}}$$

überschlagen.

Das nächste Beispiel zeigt einen positiven Fall. Bild 11 stellt den Ablauf innerhalb der ersten 14 Zeiteinheiten dar.

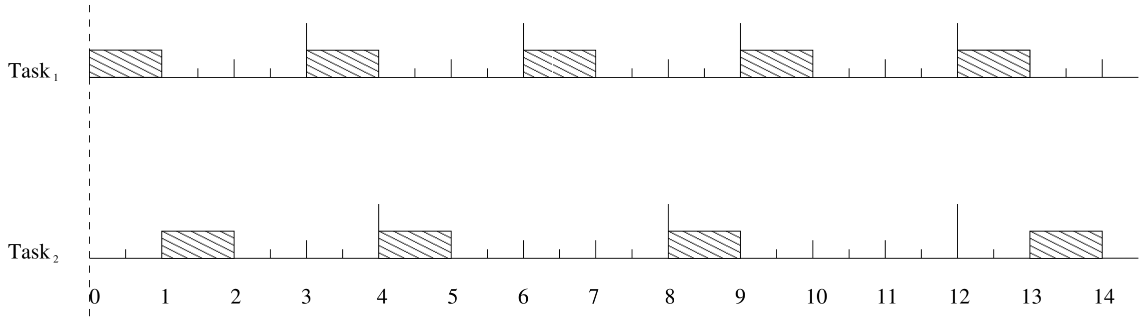

Bild 11: Laufplan der ersten 14 Zeiteinheiten für `rma_test3.dat`

Die Spezifikationsdatei für diese *tasks* sollte so aussehen:

```
# data file for the anatool.

   method: RMA
   unit: ms
   info output mode: no

   task:             1
   period:           3
   computation time: 1

   task:             2
   period:           4
   computation time: 1
```

Beispieldatei `rma_test3.dat`

Und ausgegeben wird:

```
   This set of tasks  IS    schedulable with RMA.
```

Die *tasks* im Beispiel `rma_test2.dat` sind nicht sicher planbar mit der *RM*-Laufplanmethode. Durch das ,,Planen nach Fristen" jedoch kann eine sichere Laufplanung durchgeführt werden. Hier ist die Prozessorbelastungsgrenze höher. Sie beträgt 1,0, d. h. durch die effizientere Strategie der ,,Planen nach Fristen"-Methode kann der Prozessor voll ausgelastet werden (siehe Abschnitt 3.5). Die Analyse der *tasks* aus der Spezifikationsdatei `rma_test2.dat` für die ,,Planen nach Fristen"-Methode bringt ein positives Ergebnis.

Eingabe:

```
# data file for the anatool.
info output mode: utilizations

method: EDFA
unit: ms

task:             1
period:           3
computation time: 1

task:             2
period:           4
computation time: 2
```

Beispieldatei `edfa_test1.dat`

Ausgabe:

```
Utilization of  0.833 does not exceed the utilization bound of 1.0.


This set of tasks  IS    schedulable with EDFA.
```

Außerdem werden die *tasks* auf verschiedene Weise ausgeführt. Das Verhalten der *tasks* ist im Bild 12 dargestellt.

Während zum Zeitpunkt 9 unter der ratenmonotonen Methode ein Kontextwechsel erfolgt, wird beim „Planen nach Fristen“ zunächst *task* 2 zu Ende ausgeführt. Beim „Planen nach Fristen“ werden die Prioritäten dynamisch so zugewiesen, daß immer die *task* mit der frühesten Frist die höchste Priorität bekommt und somit die andere suspendiert wird, falls eine andere gerade läuft. Am Zeitpunkt 9,0 wird *task* 1 bereit und wird nicht *task* 2 suspendieren, weil sie keine kürzere Frist hat, sondern die gleiche wie *task* 2. Somit konnte ein (mitunter teurer) Kontextwechsel gespart werden.

Das letzte Beispiel stellt eine Erweiterung eines Beispiels aus [BUT97] da. Hier werden 4 *tasks* untersucht, die Betriebsmittel benutzen und sich dadurch blockieren können. Die Spezifikation sei folgendermaßen:

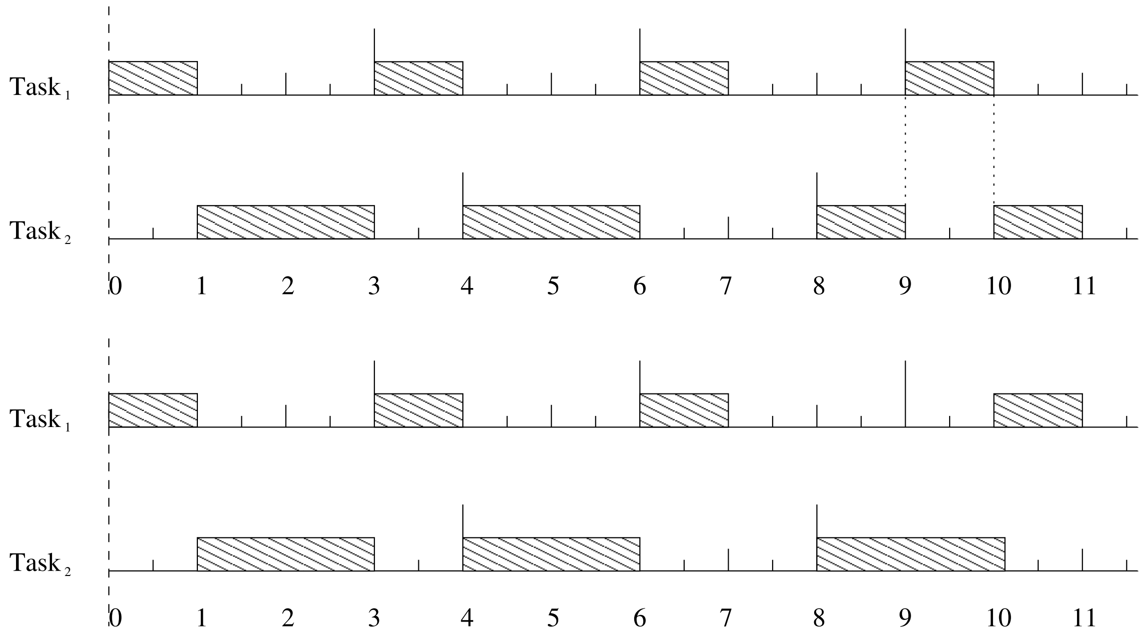

Bild 12: *RM* und *EDF* im Vergleich

```
# data file for the anatool.

method: RTA
unit: ms
resource access protocol: inheritance
info output mode: full

task:             1
period:           40
deadline:         40
computation time: 1.0, 1.0, 2.0, 2.0
resource:         {0}, {1}, {2}, {0}

task:             2
period:           120
deadline:         120
computation time: 4,   6,   5,   4,   2
resource:        {0}, {1}, {2}, {3}, {0}

task:             3
period:           80
deadline:         80
computation time: 1.0, 9.0, 3.0, 3.0
```

```
resource:          {0}, {2}, {3}, {0}

task:              4
period:            100
deadline:          100
computation time:   1,   8,   7,   3
resource:          {0}, {1}, {2}, {0}.
```

Beispieldatei `rta_test1.dat`

Von den *tasks* werden drei Betriebsmittel benutzt. Sie sind numeriert von 1 bis 3. Nur *task* 2 benutzt alle drei Betriebsmittel. Der Zugriff auf die Betriebsmittel geschieht mit dem Protokoll des Prioritätenerbens. Die anzuwendende Analysemethode ist *RTA* (siehe dazu 3.6).

Zuerst werden die *tasks* nach ihren Fristen sortiert. Somit tritt *task* 2 an letzte Stelle und erhält die Nummer 4. *Tasks* 3 und 4 rücken eins auf und bekommen die Nummern 2 und 3. Anschließend werden die Prioritäten zugewiesen, die *task* mit der kleinsten Nummer erhält die höchste Priorität.

Die Eingabe der Spezifikationsdatei `rta_test1.dat` führt zu folgender Ausgabe:

```
Analysis method: RTA
Unit: ms
Resource access protocol: INHERITANCE

                 R1[4] | R2[4] | R3[3] |
        -------------------------------
         1|      1.000   2.000   0.000
         2|      0.000   9.000   3.000
         3|      8.000   7.000   0.000
         4|      6.000   5.000   4.000

blocking factor task  1:   17.000
blocking factor task  2:   14.000
blocking factor task  3:    6.000
blocking factor task  4:    0.000

task number, now:  1; in input:  1
period: 40.000
```

```
deadline: 40.000
computation times: 1.000, 1.000, 2.000, 2.000
sum of computation times: 6.000
resources: {0}, {1}, {2}, {0}
priority:  4

task number, now:  2; in input:  3
period: 80.000
deadline: 80.000
computation times: 1.000, 9.000, 3.000, 3.000
sum of computation times: 16.000
resources: {0}, {2}, {3}, {0}
priority:  3

task number, now:  3; in input:  4
period: 100.000
deadline: 100.000
computation times: 1.000, 8.000, 7.000, 3.000
sum of computation times: 19.000
resources: {0}, {1}, {2}, {0}
priority:  2

task number, now:  4; in input:  2
period: 120.000
deadline: 120.000
computation times: 4.000, 6.000, 5.000, 4.000, 2.000
sum of computation times: 21.000
resources: {0}, {1}, {2}, {3}, {0}
priority:  1

response time task 1:   23.000
response time task 2:   36.000
response time task 3:   53.000
response time task 4:   68.000

All time data in ms.

This set of tasks  IS    schedulable with RTA.
```

Nachdem die Einstellungen über die Art und Weise der Analyse ausgegeben wurden, folgt die Tabelle, mit deren Hilfe die Blockierzeiten ausgerechnet werden (siehe auch Abschnitt 2.6.1). In dieser Tabelle stehen die längsten kritischen Abschnitte, die je *task* und Betriebmittel entstehen können. Im Beispiel gibt es je *task* und benutztem Betriebsmittel nur einen kritischen Abschnitt. Die Ausführungszeiten in diesen kritischen Abschitten werden in die Tabelle eingetragen.

```
task number, now:  1; in input:  1
computation times: 1.000,    1.000,  2.000,              2.000
resources:          {0},      {1},    {2},                {0}

task number, now:  2; in input:  3
computation times: 1.000,            9.000,  3.000,      3.000
resources:          {0},              {2},    {3},        {0}

task number, now:  3; in input:  4
computation times: 1.000,    8.000,  7.000,              3.000
resources:          {0},      {1},    {2},                {0}

task number, now:  4; in input:  2
computation times: 4.000,    6.000,  5.000,  4.000,      2.000
resources:          {0},      {1},    {2},    {3},        {0}

                         R1[4] | R2[4] | R3[3] |
                    -------------------------------
                     1|      1.000   2.000   0.000
                     2|      0.000   9.000   3.000
                     3|      8.000   7.000   0.000
                     4|      6.000   5.000   4.000
```

Die Zeilen sind mit den *task*-Nummern bezeichnet und die Spalten mit der Nummer des Betriebsmittels sowie der zugehörigen Grenzprioritäten (siehe Abschnitt 2.6.1). Spalte 1 enthält demnach die Ausführungszeiten der längsten kritischen Abschnitte des Betriebsmittels R_1, in die die *tasks* 1, 3, und 4 eintreten können, wobei Betriebsmittel 1 als Grenzpriorität die Priorität von *task* 4 hat (`R1[4]`).

Die Tabelle wird in der Unterprozedur `Make_Table` erstellt. Dort werden auch die Grenzprioritäten für die Betriebsmittel errechnet. Als Datenstruktur wird eine zweidimensionale Reihung namens `Task_Resource_Tab` vereinbart.

Aus der Tabelle die Blockierzeiten abzuleiten ist nicht trivial. Nähere Erklärungen dazu sind in Abschnitt 2.6.1 nachzulesen. Hier nur soviel:

Am Beispiel der *task* 1 soll gezeigt werden, wie sich die Blockierzeiten errechnen.

Die Blockierzeit ergibt sich aus:

$$B_1 = min(B_1^l, B_1^s).$$

Dabei sind

$$B_1^l = \sum_{j=1+1}^{4} \max_k [D_{j,k} : C(S_k) \geq P_1]$$

und

$$B_1^s = \sum_{k=1}^{3} \max_{j>1} [D_{j,k} : C(S_k) \geq P_1].$$

Der Index j indiziert alle *tasks* mit niedrigerer Priorität als die der *task* τ_i ($\tau_2 ... \tau_4$). k indiziert alle Betriebsmittel, hier: $k = 1...3$.

B_1^l ist demnach die Summe der Ausführungszeiten der längsten kritischen Abschnitte niederpriorisierter *tasks* die τ_1 blockieren können, die also Betriebsmittel benutzen, die auch τ_1 benutzt. Für das aktuelle Beispiel sind das:

	R1[4]	R2[4]	R3[3]
1	1.000	2.000	0.000
2	0.000	$\underline{9.000}$	3.000
3	$\underline{8.000}$	7.000	0.000
4	$\underline{6.000}$	5.000	4.000

Also ist $B_1^l = 9,0 + 8,0 + 6,0 = 23$.

B_1^s ist die Summe der Ausführungszeiten der längsten kritischen Abschnitte, in denen ein Betriebsmittel gehalten werden kann und zwar von den *tasks* $\tau_2 ... \tau_4$. In der Tabelle sind diese Werte unterstrichen:

	R1[4]	R2[4]	R3[3]
1	1.000	2.000	0.000
2	0.000	$\underline{9.000}$	3.000
3	$\underline{8.000}$	7.000	0.000
4	6.000	5.000	4.000

Betriebsmittel R_3 wird von τ_1 nicht benutzt. Deshalb kann τ_1 auch nicht an R_3 blockiert werden. Somit bleibt R_3 außer acht.

Nun ist $B_1^s = 9,0 + 8,0 = 17,0$.

Nach Anwendung der Formel

$$B_1 = min(B_1^l, B_1^s)$$

ergibt sich

$$B_1 = min(23, 17) = 17.$$

Hiermit haben wir die Blockierzeit B_1 für *task* τ_1.

Die Blockierzeiten für die *tasks* τ_2 und τ_3 werden auf die gleiche Weise berechnet. Für *Task* τ_4 existiert keine Blockierzeit, weil es keine niederpriorisierte *task* gibt, die τ_4 an irgendeinem Betriebsmittel blockieren könnte.

Sind alle Blockierzeiten ausgerechnet (mit der Unterprozedur `Get_Blocking_Times`), dann werden die Antwortzeiten für $\tau_1 ... \tau_4$ berechnet. Das soll hier am Beispiel von τ_3 durchgespielt werden.

Aus der allgemeinen Formel für die Berechnug der Antwortzeit einer *task* (S. 38) wird im Falle der Berechnung von τ_3 folgende, speziellere Formel:

$$R_3 = C_3 + B_3 + \sum_{j=1}^{3-1} \lceil \frac{R_3}{T_j} \rceil C_j.$$

Die iterative Berechnung der Antwortzeit R_3 geht wie folgt:

$$
\begin{aligned}
R_3^1 &= C_3 + B_3 = 19 + 6 = 25 \\
R_3^2 &= C_3 + B_3 + \sum_{j=1}^{3-1} \lceil \frac{R_3^1}{T_j} \rceil C_j \\
&= 19 + 6 + \lceil \frac{25}{40} \rceil 6 + \lceil \frac{25}{100} \rceil 16 \\
&= 19 + 6 + 1 \cdot 6 + 1 \cdot 16 \\
&= 19 + 6 + 22 \\
&= 47 \\
R_3^3 &= 19 + 6 + \lceil \frac{47}{40} \rceil 6 + \lceil \frac{47}{100} \rceil 16 \\
&= 19 + 6 + 2 \cdot 6 + 1 \cdot 16 \\
&= 19 + 6 + 28 \\
&= 53 \\
R_3^4 &= 19 + 6 + \lceil \frac{53}{40} \rceil 6 + \lceil \frac{53}{100} \rceil 16 \\
&= 19 + 6 + 2 \cdot 6 + 1 \cdot 16 \\
&= 19 + 6 + 28 \\
&= 53 \\
53 &= 53
\end{aligned}
$$

Weil $R_3^4 = R_3^3$ ($R_i^{k+1} = R_i^k$), ist die Berechnung abgeschlossen. Da $R_3 < C_3$ ($53 < 100$), haben wir eine Antwortzeit kleiner als die Frist, womit die *task* τ_3 planbar ist. Und weil auch die anderen *tasks* Antwortzeiten haben, die kleiner sind als ihre Fristen, könnte die untersuchte Menge der *tasks* die in `rta_test1.dat` spezifiziert wurden, in einem Echtzeitsystem eingesetzt werden.

9 Ergebnisse dieser Arbeit

Mit der vorliegenden Arbeit wurde eine gute Grundlage für ein Werkzeug geschaffen, daß vielleicht schon bald Entwicklern bei der Planung und beim Entwurf von Echtzeitsystemen helfen wird, die Laufeigenschaften der *tasks* der Systeme vor Inbetriebnahme zu erkennen.

Durch eine Gegenüberstellung der beschriebenen Planbarkeitsanalysemethoden konnten verschiedene Vor- und Nachteile beschrieben werden. Damit ist es möglich, gezielter die eine oder andere Methode auszuwählen. Der Planbarkeitsanalysator soll bei der Auswahl eine Hilfestellung geben.

Bei der Betrachtung des Gebietes der Planbarkeitstheorie stellte sich heraus, daß dort noch einiges getan werden muß, um die Ergebnisse der Theorien auch in die Praxis umsetzen zu können.

Es sollte erstrebenswert sein, Entwicklungsumgebungen zu schaffen, mit denen nicht nur die algorithmischen, sondern auch die zeitlichen Zusammenhänge des geplanten Systems untersucht werden können. Das das möglich ist, konnte bei der Erstellung dieser Arbeit erkannt werden.

10 Ausblick

Außer dem Planbarkeitsanalysator werden im Projekt ADAM noch andere Werkzeuge entwickelt. Diese sollen zu einem System zusammengefügt werden, mit dem eine Zeitanalyse von Ada-Anwendungen durchführbar ist. Mit dieser Analyse wird das zeitliche Verhalten von Ada-Programmen vor dem Einsatz in Echtzeitumgebungen untersucht. Damit kann schon in der Planungsphase der Anwendung festgestellt werden, ob die Anwendung später in der Lage sein wird, die gestellten Zeitanforderungen einzuhalten.

In diesem Analysesystem wird der Planbarkeitsanalysator das letzte Glied sein. Die von den anderen Komponenten berechneten Werte werden als Eingangsgrößen für die Planbarkeitsanalyse bereitgestellt.

Bisher werden keine Verzögerungen seitens des Betriebssystems vom Planbarkeitsanalysator berücksichtigt. Es soll zukünftig möglich sein, Verzögerungszeiten von auftretenden Unterbrechungen (engl. *interrupts*), Kontextwechseln und *cache*-Einflüssen in die Analyse einzubringen. Dazu sind veränderte Laufplananalysemethoden notwendig. Zum Teil müssen diese erst entwickelt werden, wie beispielsweise die Analyse für das Planen nach Fristen unter Einbeziehung von *cache*-Einflüssen.

Momentan nutzt der Planbarkeitsanalysator die in der Eingabe eingestellten Methoden, um die zu untersuchenden *tasks* auf Planbarkeit zu überprüfen. Das soll dahingehend geändert werden, daß die zu untersuchende *task*-Menge mit allen Methoden analysiert wird und die Ergebnisse dieser Analysen im Vergleich ausgegeben werden.

Es ist auch denkbar, die Ergebnisse der Planbarkeitsanalyse graphisch ausgeben zu lassen, um sie dem Benutzer anschaulicher zu präsentieren. Dazu wäre eine Schnittstelle zu einer graphikerzeugenden Komponente nötig, die beispielsweise in der Programmiersprache JAVA entwickelt sein könnte.

Erklärung zur Urheberschaft

Hiermit erkläre ich, daß die vorliegende Arbeit allein durch meine Arbeit entstanden ist. Außer der angegebenen habe ich keine Literatur benutzt, die Informationen über den Gegenstand der Arbeit beinhaltet.

Oliver Bühn

Literatur

[AARM] o. V.: *Annotated Ada Reference Manual.* Version 6.0. Intermetrics, Inc. 1994

[BAK98] Baker, T. P.: *Real-Time Systems.* Folien zur Vorlesung.

[BUCH] Buchholz, Bernhard: *Prozeßdatenverarbeitung.* Skript zur Vorlesung. Version 2.0

[BARN98] Barnes, John: *Programming in Ada 95.* 2. Auflage. Harlow: Addison Wesley Longman Limited 1998

[BUT97] Buttazzo, Giorgio C.: *Hard real-time computing systems.* Norwell, MA: Kluwer Academic Publishers 1997

[HORN98] Horn, Christian (Hrsg.) und Kerner, Immo O. (Hrsg.): *Lehr- und Übungsbuch Informatik.* Band 4: *Technische Informatik und Systemgestaltung.* München, Wien: Fachbuchverlag Leipzig im Carl Hansa Verlag 1998

[KAST90] Kastens, Uwe: *Handbuch der Informatik.* Band 3.3: *Übersetzerbau.* München: R. Oldenbourg Verlag 1990

[LL73] Liu, C. L. und Layland, James W.: *Scheduling algorithms for multiprogramming in a hard real-time environment.* In: Journal of the association for computing machinery, 46...61, Januar 1973

[LIU97] Liu, Jane: *real-time systems.* Manuskript. Oktober 1997

[SRL90] Sha, l; Rajkumar, R.; Lehoczky, J. P.: *Priority inheritance protocols in multiprocessor real-time systems.* In: IEEE Transactions on Computers, H. 39, September 1990

[TAN95] Tanenbaum, Andrew S.: *Distributed operating systems.* Englewood Cliffs, N. J.: Prentice Hall, Inc. 1995

Stichwortverzeichnis

Ada, 24
ADAM, 19, 21, 73
Aktivierungszeit, 12, 27
Antwortzeit, 13, 27, 56
Aufgabe, 46
Ausführungszeit, 12
Ausnahmebehandlung, 57
Ausschluß, gegenseitiger, 22, 25

Backus-Naur-Form, 47
Beendigungszeit, 27
Betriebsmittel, 17, 21, 28
  -zugriffsprotokoll, 28, 38
Blockierzeit, 24, 45
Blockierzeiten, 35
BNF, *siehe* Backus-Naur-Form

cache, 19
  -Einflüsse, 19
  -Simulator, 19

Echtzeit, 9
  -bedingung, 9
  -betriebssystem, 11
  -system, 10
EDF, *siehe* Planbarkeitsanalyse
EDFA, *siehe* Planbarkeitsanalyse
Eingabeschnittstelle, 47

Frist, 12, 14
fristenmonotone Laufplanmethode, 38

Grenzpriorität, 68

Instanz, 17

Kenngrößen von tasks, 16
Kontext, 47
Kontextwechsel, 19
Kontrollfaden, 11
kritische Zeitzone, 27, 53
kritischer Abschnitt, 22, 29, 32, 39
kritischer Moment, 27

Laufplaner, 14
Laufplanung, 14
Laufplanungsmethode, 14

multitasking, 11

Nebenläufigkeit, 12

Periode, 12, 17
pipelining, 20
  -Simulator, 21
Planbarkeitsanalysator, 28, 46
  Ausgabe, 43, 48, 58
  Benutzung, 57
  Eingabe, 39, 50
  Programmiersprache, 48
Planbarkeitsanalyse, 16
  -methode, 16
  Analyse der Antwortzeiten, RTA, 16, 56, 66
  Analyse der Antwortzeiten, 37
  erweiterte ratenmonotone, 35, 38
  Planen nach Fristen, EDFA, 36
  Probleme, 18
  ratenmonotone, 16, 33
Planbarkeitstheorie, 17, 73
Planen nach Fristen, 36, 63
Priorität, 13
Prioritätenumkehrung, 28
Prozeß, 11
Prozessorbelastung, 26, 33
Prozessorbelastungsgrenze, 26, 35, 61
Pseudoparallelität, 12

Rendezvous, 24
Ressource, 21
RMA, 16
RTA, *siehe* Planbarkeitsanalyse

scheduler, *siehe* Laufplaner

scheduling, *siehe* Laufplanung
Spezifikationsdatei, 40, 48
Synchronisation, 24

task, 11, 17
    nichtperiodische, 13, 14, 25
    periodische, 13, 25
thread, 11

unkritischer Abschnitt, 23

verdrängend, 14

Warteschlange, 16
Wichtigkeit, 22

Zugriffsprotokoll für Betriebsmittel, 43, 45
    Grenzpriorität, 31
    Prioritätenerben, 45, 66