Energieeinsparung durch kooperative Gerätenutzung

Studienarbeit im Fach Informatik

vorgelegt von

Steffen Meyer

geboren am 12. Mai 1976 in Nürnberg

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich−Alexander−Universität Erlangen−Nürnberg

Betreuer:	*Prof. Dr. rer. nat. Fridolin Hofmann* *Dr. Ing. Frank Bellosa*
Beginn der Arbeit:	*16. Oktober 2000*
Abgabe der Arbeit:	*07. Februar 2001*

Inhaltsverzeichnis

1. Einleitung

Im Bereich der mobilen Systeme hat in den letzten Jahren eine enorme Weiterentwicklung stattgefunden. Notebooks sind schneller geworden, haben hochauflösende Bildschirme und eine Festplattenkapazität, die noch vor wenigen Jahren nur in Fileservern zu finden war. Viele Zusatzgeräte, wie CD−ROM und DVD Laufwerke oder integrierte Modems und ISDN−Karten gehören heute bereits zum Standard, andere Erweiterungen, wie aus dem Bereich der drahtlosen Kommunikation, werden immer verbreiteter. Das alles hat einen stark gestiegenen Energieverbrauch zur Folge. Weiterentwicklungen beim Bau von Akkus allein können dies nicht auffangen. Vielmehr ist es notwendig, dass jede einzelne Komponente ihren Verbrauch einschränkt, wo immer es möglich ist.

Um bei mobilen Systemen Energie einzusparen und damit die Nutzungsdauer im vom Stromnetz unabhängigen Betrieb zu erhöhen, besitzen moderne Festplatten mehrere Betriebsmodi in denen sie unterschiedlich viel Leistung aufnehmen. Dies wird möglich, indem der Antrieb und Teile der Elektronik mit verminderter Leistung betrieben oder ganz abgeschaltet werden. Die niedrigere Leistungsaufnahme wird somit durch verminderte Leistungsfähigkeit erkauft. Es ist üblich, die Festplatte in einen Sparmodus zu versetzen, falls sie über einen gewissen Zeitraum nicht verwendet wird. Diese Zeit wird Idle−Time genannt.

Bei der Entscheidung über den Moduswechsel muss berücksichtigt werden, dass dieser zusätzlich Energie verbraucht. Ist die erwartete Idle−Time sehr kurz, kann es sinnvoller sein, im derzeitigen Energiemodus zu bleiben. Die Idle−Time im voraus zu bestimmen, ist in interaktiven Systemen nicht möglich. Daher beschäftigt sich eine Vielzahl von Algorithmen zur Energieeinsparung damit, aus einer Analyse der bisher erfolgten Zugriffe verlässliche Prognosen für zukünftige Aktivitäten zu erstellen. Die Ansätze reichen von festen oder dynamisch angepassten Timeouts über statistische Verfahren bis hin zu lernenden Algorithmen (vergleiche [LM99], sowie [LCS00]). Ziel ist es, aussagekräftige Kriterien, für das Wechseln in einen Sparmodus zu finden. Befindet sich die Festplatte im Sparmodus, veranlasst der nächste Zugriff sofort den Spinup. Es besteht oft keine Möglichkeit, weniger wichtige Aktionen zu verzögern, um damit das sofortige Wiederanlaufen der Platte zu verhindern. [LBM00] stellt einen Konzept vor, das diese Idee aufgreift. Ein erweiterter Scheduling−Algorithmus bezieht energetische Überlegungen mit ein, so dass Prozesse, die auf die Festplatte zugreifen wollen, möglicherweise erst verspätet lauffähig werden. Der betroffene Prozess hat darauf jedoch keinen Einfluss.

In dieser Arbeit soll ein Ansatz verfolgt werden, bei dem die Prozesse die Dringlichkeit ihrer Aktionen selbst einstufen. Die Energieeinsparung soll durch kooperative Nutzung der Festplatte erfolgen. Die Grundidee dabei ist, das marktwirtschaftliche Prinzip von Angebot und Nachfrage zu übertragen. Es soll ein Marktplatz geschaffen werden, auf dem Prozesse Energie anbieten, um Leistungen vom Betriebssystem zu erhalten. Jeder Prozess verfügt über ein Energiekonto, von dem ausgegebene (d.h. von ihm verbrauchte) Energie abgezogen wird. Will ein Prozess eine Aktion ausführen, die Energie benötigt, so bietet er dem Betriebssystem einen gewissen Energiewert an. Das Betriebssystem entscheidet, ob es wirtschaftlicher ist, die Aktion sofort auszuführen, oder diese zu verzögern, bis andere Prozesse ebenfalls Aktionen ausführen wollen. Wirtschaftlichkeit hängt davon ab, ob sich die Festplatte im Active−Modus befindet und wie teuer ein Wechsel in diesen Modus ist. Auf diese Weise können sich die Gebote mehrerer Prozesse ansammeln, so dass in der Summe genug Energie zusammenkommen kann, um die geforderten Aktionen wirtschaftlich ausführen zu können.

Ziel der Arbeit ist es, diese Idee zu einem Konzept zu konkretisieren und dieses zu implementieren. Dadurch können Prozesse den Energieverbrauch ihrer Festplattenoperationen selbst beschränken. Im praktischen Teil soll ein MINIX−System um die notwendigen Komponenten der Energieverwaltung erweitert werden.

Um den vorgegebenen Verbrauch tatsächlich einhalten zu können, muss der Energieverbrauch der Festplatte in den einzelnen Modi, sowie die Kosten eines Übergangs zwischen den Modi möglichst genau bekannt sein. In Kapitel 2 werden daher die energetischen Charakteristika einer speziellen Festplatte eingehend untersucht und es wird geklärt, in welchem Rahmen Einsparungen möglich sind. Dazu sind umfangreiche Messungen durchgeführt worden.

In Kapitel 3 wird das Konzept der kooperativen Nutzung theoretisch entwickelt. Es werden die benötigten Komponenten vorgestellt und ihre Aufgaben erläutert. Abschließend wird das Konzept kritisch hinterfragt.

Kapitel 4 zeigt die Einbettung in ein konkretes System. Hierzu wurde MINIX als Platform gewählt. Dabei steht zuerst die Beschreibung der Implementierung und der dabei aufgetretenen Probleme im Vordergrund. Anschließend werden experimentell Parameter ermittelt, um das System realen Gegebenheiten anzupassen. Ausführliche Tests dienen als Bewertungsgrundlage und lassen einen Vergleich mit einem konventionellen Algorithmus zu.

Kapitel 5 gibt letztlich Anstöße, wie durch weitere Arbeiten, die hier zu weit geführt hätten, die Resultate verbessert werden können.

Abschließend werden die Ergebnisse kurz zusammengefasst.

2. Energieeinsparung bei Festplatten

Die Anforderung, mit mobilen Systemen möglichst lange unabhängig vom Stromnetz arbeiten zu können, zwingt die Hersteller, auch bei Peripheriegeräten wie Festplatten auf eine geringe Leistungsaufnahme zu achten.

Fast alle Festplatten bieten daher heute gewisse Konzepte zur Energieeinsparung. Meist wird eine Reihe von Modi zur Verfügung gestellt. Die Grundidee dabei ist, geringere Leistungsaufnahme durch verminderte Leistungsfähigkeit zu erreichen. Da Datenaustausch in der Regel nicht kontinuierlich erfolgt, sondern öfters über längere Perioden kein Zugriff notwendig ist, liegt es nahe, nicht benötigte Bauteile und Schaltkreise zwischenzeitlich abzuschalten. Dies muss allerdings durch eine höhere Reaktionszeit erkauft werden, da dann im Falle einer Anforderung die abgeschalteten Komponenten erst wieder hochgefahren werden müssen.

Im folgenden soll ein Überblick über die gängigen Energiesparmodi gegeben werden.

2.1. Sparmodi moderner Festplatten

2.1.1. Überblick über die Modi

Eine typische Festplatte, wie sie in modernen Notebooks zum Einsatz kommt, unterstützt nach [IBM99] vier Modi:

- Active Mode
- Idle Mode
- Standby Mode
- Sleep Mode

Im Active Mode werden Daten gelesen oder geschrieben, die Platte positioniert und bearbeitet Kommandos. Es können ohne Verzögerung Anforderungen ausgeführt werden. Üblich ist ein Verbrauch zwischen 2,0 und 2,5 Watt.

Im Idle Mode werden Teile der Elektronik abgeschaltet. Oft wird die Positionierungsüberwachung und Lese−/Schreibelektronik deaktiviert, teilweise werden die Köpfe in Parkposition gefahren. Die Platte dreht aber weiterhin und nimmt Kommandos entgegen. Ein eintreffendes Kommando veranlasst die Platte in den Active Modus zu wechseln. Dabei ist mit einer Verzögerung im Bereich von 40 ms zu rechnen. Der Energieverbrauch in diesem Modus sinkt auf weniger als 1 Watt.

Im Standby Mode ist der Spindel−Motor angehalten. Ein großer Teil der Elektronik ist abgeschaltet. Die Platte nimmt nach wie vor Kommandos entgegen und wechselt gegebenenfalls in den Active Mode. Die dabei auftretende Verzögerung liegt bei Notebookplatten im Bereich von 1,5 bis 5 Sekunden. Verglichen mit Platten für Desktop−PCs, bei denen bis zu 30 Sekunden Verzögerung keine Seltenheit ist, ist dieser Wert sehr niedrig. Der Energieverbrauch liegt bei ungefähr 0,3 Watt.

In den Sleep Mode sollte nur gewechselt werden, wenn die Platte lange Zeit nicht gebraucht wird (mehrere Stunden). Nahezu die gesamte Elektronik ist abgeschaltet, nur auf ein Kommando zum

Aufwecken (Reset) kann noch reagiert werden. Die Verzögerung beim Wechsel in den Active Mode beträgt einige Sekunden. Der Energieverbrauch sinkt im Sleep Mode auf etwa 0,1 Watt.

2.1.2. Wechsel zwischen den Modi

Grundsätzlich wird nur in einen anderen Modus gewechselt, wenn

- die platteneigene Energiesparstrategie dies verlangt.
- ein eintreffendes Kommando den Wechsel in einen anderen Modus forciert.

Der Sleep Mode bildet dabei eine Ausnahme. Er muss durch ein spezielles Kommando eingeleitet werden und kann nur durch ein Reset−Kommando wieder verlassen werden. Der langen Verzögerung beim Wechsel in den Active Mode steht ein eher geringer Einspareffekt (verglichen mit dem Standby Mode) gegenüber. Deshalb wird er in Energiesparkonzepten meist nicht berücksichtigt.

Der Wechsel zwischen den Modi Active, Idle und Standby wird von der Energiesparstrategie der Festplatte selbst bestimmt, soweit keine anderen Kommandos eintreffen.

Der übliche Ansatz dabei ist folgender:

Befindet sich die Platte im Active Mode, so wird nach Abarbeitung des letzten Kommandos eine feste Zeitspanne (Idle−Timeout, üblich sind 5 Sekunden) gewartet. Trifft kein weiteres Kommando ein, so wird in den Idle Mode gewechselt. Der Übergang zum Standby Mode wird ähnlich geregelt, nur die Wartedauer ist länger. Treffen im Idle Mode für 5 Minuten (Standby−Timeout) keine Kommandos ein, so geht die Platte in den Standby Mode. Ein eintreffendes Lese−, Schreib− oder Positionierungskommando bewirkt jeweils den Wechsel in den Active Mode. Folgender Graph verdeutlicht nochmals diese Strategie:

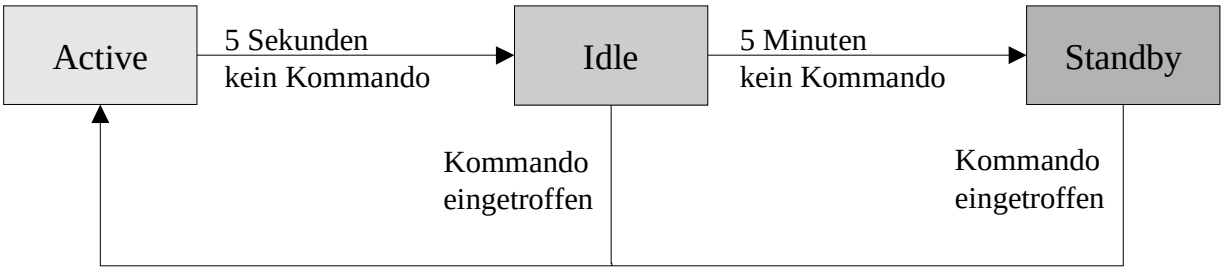

Abbildung 1: Konventionelle Energiesparstrategie bei Festplatten

Diese Strategie ist nicht optimal, wie leicht einzusehen ist. Sie geht davon aus, dass wenn über eine gewisse Zeitspanne kein Zugriff auf die Festplatte erfolgt ist, auch in nächster Zukunft keiner erfolgen wird. Diese Annahme ist jedoch oft nicht richtig. So trifft sie zwar zu, wenn der Benutzer seinen Arbeitsplatz gerade kurzzeitig verlassen hat, in vielen anderen Fällen jedoch läuft dieser Ansatz ins Leere. Stellt man sich vor, eine Anwendung greift periodisch auf die Platte zu, in Abständen, die etwas größer sind als der Standby−Timeout, so wird die Platte stets nur kurz in den Standby Mode springen und danach sofort wieder heraus. Das kann dazu führen, dass Energie verschwendet statt eingespart wird, da jeder Wechsel zwischen den Modi auch Energie kostet (siehe dazu Kapitel 2.2.2).

Das System, in dem die Platte betrieben wird, kann nur zu einem kleinen Teil in die obige Strategie eingreifen. Zwar kann der Standby−Timeout in einem gewissen Rahmen konfiguriert werden, der Idle−Timeout bleibt aber fest. Der einzige Ausweg ist daher, die Energiesparstrategie der Festplatte durch einen sinnlos langen Standby−Timeout teilweise auszuhebeln und stattdessen den Wechsel in den Standby Mode nach einer ins Betriebssystem eingebauten, besseren Strategie mittels expliziter Wechselkommandos zu steuern. Dieser Weg wurde auch in dieser Arbeit beschritten.

2.2. Energieverbrauch der IBM DTNA−22160

In dem für die Implementierung eingesetzten Testsystem wird eine Platte der Firma IBM vom Typ DTNA−22160 verwendet. Es handelt sich dabei um eine Notebook−Festplatte im 2,5 Zoll Format, die für den Einsatz in Geräten der IBM Thinkpad−Serie konzipiert wurde. Die Platte besitzt laut Spezifikation [IBM97] folgende Kennwerte:

Interface	ATA −3
Formatted capacity	2.1 GB
Rotation speed	4000 rpm
Data transfer rate	16.6 MB/s
Avarage reed seek time	13 ms
Data buffer size	96 KB
Number of disks	3
Number of heads	6

Tabelle 1: Kennwerte der IBM DTNA−22160

Es werden die bekannten vier Betriebsmodi Active, Idle, Standby und Sleep unterstützt. Der Idle Mode ist allerdings aufgeteilt in einen Performance Idle Mode und einen Low Power Idle Mode.

Sofort nachdem Lese−/Schreib− oder Positionierungsoperationen beendet sind, wird aus dem Active Mode in den Performance Idle Mode gewechselt. Treffen weitere Kommandos ein, so können sie ohne Verzögerung bearbeitet werden. Lediglich ein kleiner Teil der Elektronik ist abgeschaltet. Die Zeit, die in diesem Modus verbracht wird bis die Platte Low Power Idle Mode wechselt, wird laut [IBM97] durch einen platteneigenen Algorithmus selbst bestimmt. Meinen Messungen zufolge liegt sie aber immer im Bereich von 1 bis 2 Sekunden.

Im Low Power Idle Mode werden weitere Teile der Elektronik abgeschaltet . Die Verzögerung und Energieaufnahme liegt im Bereich des konventionellen Idle Mode.

Um einen genauen Überblick über die Energieaufnahme dieser speziellen Platte als Basis für weiterführende Überlegungen zu bekommen, wurden umfangreiche Messungen durchgeführt und diese mit den Werten der Spezifikation verglichen. Details zur Durchführung der Messungen sind im Anhang A beschrieben. Die Ergebnisse sollen hier dargestellt werden.

2.2.1. Verbrauch in den einzelnen Energiemodi

In der Spezifikation der Platte sind folgende Werte als maximale Leistungsaufnahme angegeben.

Energiemodus	***Max. Leistungs−aufnahme in Watt***
Active	2,3
Performance Idle	1,85
Low Performance Idle	0,85
Standby	0,3
Sleep	0,1

Tabelle 2: Maximale Leistungsaufname der IBM DTNA−22160 laut [IBM97]

Da es sich hierbei um Maximalwerte handelt, wurden eigene Messungen durchgeführt, um Durchschnittswerte für den Verbrauch im realen System zu bekommen. Folgende Tabelle gibt einen Überblick über die gemessenen Werte.

Energiemodus	***Leistungsaufnahme in Watt***
Active	3,29
Performance Idle	2,49
Low Performance Idle	1,15
Standby	0,27
Sleep	0,08

Tabelle 3: Gemessene Leistungsaufnahme der IBM DTNA−22160 in Watt

Vergleicht man die gemessenen Werte mit den Sollwerten, so fällt auf, dass im Active, Performance Idle und Low Performance Idle Mode der gemessene Verbrauch über dem Maximalwert laut Spezifikation liegt. Der Grund dafür ist wohl in der Messanordnung zu suchen, da stets die Leistungsaufnahme des Gesamtsystem gemessen wird. Somit schlägt sich der Mehrverbrauch anderer Komponenten, wie des Festplattenkontrollers, der ja in den „aktiveren“ Modi ebenfalls mehr Leistung aufnimmt, in den Messwerten nieder. Da es sich hierbei um den tatsächlichen Verbrauch des Systems handelt, werden im Folgenden die gemessenen Werte statt der reinen Verbrauchswerte der Festplatte als Grundlage für weitere Überlegungen verwendet.

2.2.2. Kosten beim Wechsel der Energiemodi

Mindestens genauso wichtig, wie der Verbrauch in den einzelnen Energiemodi, sind die Kosten, die beim Wechsel entstehen. Da der Spezifikation des Herstellers keine Angaben zu entnehmen sind, kommt den experimentell ermittelten Werten hier eine besondere Bedeutung zu.

Bevor der Energieverbrauch im Detail untersucht wird, soll kurz auf die Bedingungen eingegangen werden, unter denen ein Wechsel in einen anderen Energiemodus erfolgt. Folgende Tabelle gibt einen Überblick, wann in einen sparsameren Modus gewechselt wird.

Wechsel von	***nach***	***Von der Platte iniziiert?***	***Vom Hostsystem beeinflussbar?***
Active	Performance Idle	Ja. (sofort nach Abschluss des letzten Kommandos)	Nein.
Performance Idle	Low Performance Idle	Ja. (nach Ablauf eines Ti−meouts, Zeit wird bestimmt durch platten−eigenen Algorithmus, meist 1 bis 2 Sekunden)	Nein.
Low Performance Idle	Standby	Ja. (nach Ablauf des Standby−Timeouts)	Ja. (Timeout ist konfigurier−bar, zusätzlich explizites Wechselkommando vor−handen)
Standby	Sleep	Nein.	Ja. (explizites Wechsel−kommando vorhanden)

Tabelle 4: Bedingungen für den Wechsel in einen niedrigeren Energiemodus

Es fällt auf, dass nur der Wechsel in den Standby Mode vom Hostsystem beeinflussbar ist. Daraus folgt, dass sich betriebsystemeigene Energiesparstrategien darauf beschränken müssen, Entscheidungen über einen Wechsel in diesen Modus zu fällen. Der Sleep Mode wird hier aus bereits genannten Gründen nicht berücksichtigt, er ist hier nur der Vollständigkeit halber aufgeführt.

Beim Wechsel in einen höheren Energiemodus gibt es folgende Fälle:

Wechsel von	*nach*	*Bedingung*
Performance Idle, Low Performance Idle, Standby	Active	Lese−, Schreib− oder Positionierungs−kommando trifft ein
Sleep	Active	Resetkommando trifft ein
Low Performance Idle, Standby	Performance Idle	Explizites Wechselkommando trifft ein

Tabelle 5: Bedingungen für den Wechsel in einen höheren Energiemodus

Der Standardfall besteht darin, dass ein eintreffendes Lese−, Schreib− oder Positionierungskommando die Platte veranlasst, in den Active Mode zu wechseln (Fall 1). Eine betriebsystemeigene Energiesparstrategie muss also entscheiden, wann das Ausführen eines solchen Kommandos und das damit verbundene Hochfahren der Platte notwendig und energetisch vertretbar ist. Ein Wechsel in den Performance Idle Mode durch ein explizites Wechselkommando (Fall 3) kann dagegen außer Acht gelassen werden, da es im Allgemeinen wenig sinnvoll ist, in einen höheren Energiemodus zu wechseln, wenn nicht auch Aktionen getätigt werden sollen.

Zusammenfassend ist zu sagen, dass nur einige wenige Wechselszenarien von praktischem Interesse sind und deshalb energetisch untersucht werden müssen. Prinzipiell müssen Wechsel, die das Hostsystem nicht beeinflussen kann, nicht separat untersucht werden. Sie sind vielmehr als zwangsläufige Folge eines vorhergegangenen Wechsels zu sehen und bei diesem mit zu betrachten. Diese Übergänge zwischen den Energiemodi sind somit von Bedeutung:

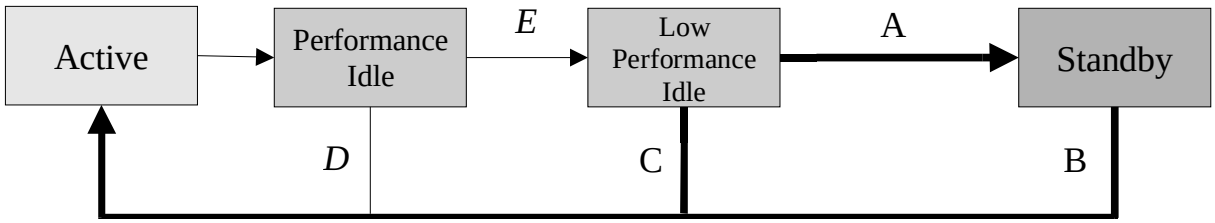

Abbildung 2: Wichtige Übergänge zwischen den Energiemodi

Die Fälle A, B und C treten am häufigsten auf und werden im Folgenden genauer untersucht. Der Übergang von Active nach Performance Idle erfolgt sofort nach Ausführung des Kommandos, so dass dieser bei einem Wechsel in den Active Mode als Folge mitbetrachtet wird.

Beim Fall D ergibt sich ein Problem. Das Hostsystem kann über das Interface der Platte nur feststellen, ob sich die Platte im Standby Mode befindet oder nicht. Damit ist es möglich den Fall B zu erkennen, nicht aber C und D zu unterscheiden. Da der Wechsel E, wie bereits angeführt, nicht beeinflussbar ist und auch keine feste Zeitspanne existiert, nach der dieser Wechsel auftritt, muss hier eine vereinfachte Sicht gewählt werden. Eine Möglichkeit ist es, davon auszugehen, dass Fall D nicht auftritt, sondern nur Fall C. Dieser Ansatz schätzt die verbrauchte Energie nach oben ab und wird nachfolgend verwendet, um die Betrachtungen zu vereinfachen. Der Wechsel von Performance Idle nach Low Performance Idle wird somit ebenfalls als Folge eines Wechsels in den Active Mode gesehen.

Fall A: Wechsel von Low Performance Idle nach Standby

Es wurde eine Messreihe von 10 Einzelmessungen angelegt. Da sich die gewonnenen Messkurven nur minimal unterscheiden, wurden sie als realistisch angesehen und gemittelt. Dabei ist die in Abbildung 3 gezeigte Kurve entstanden.

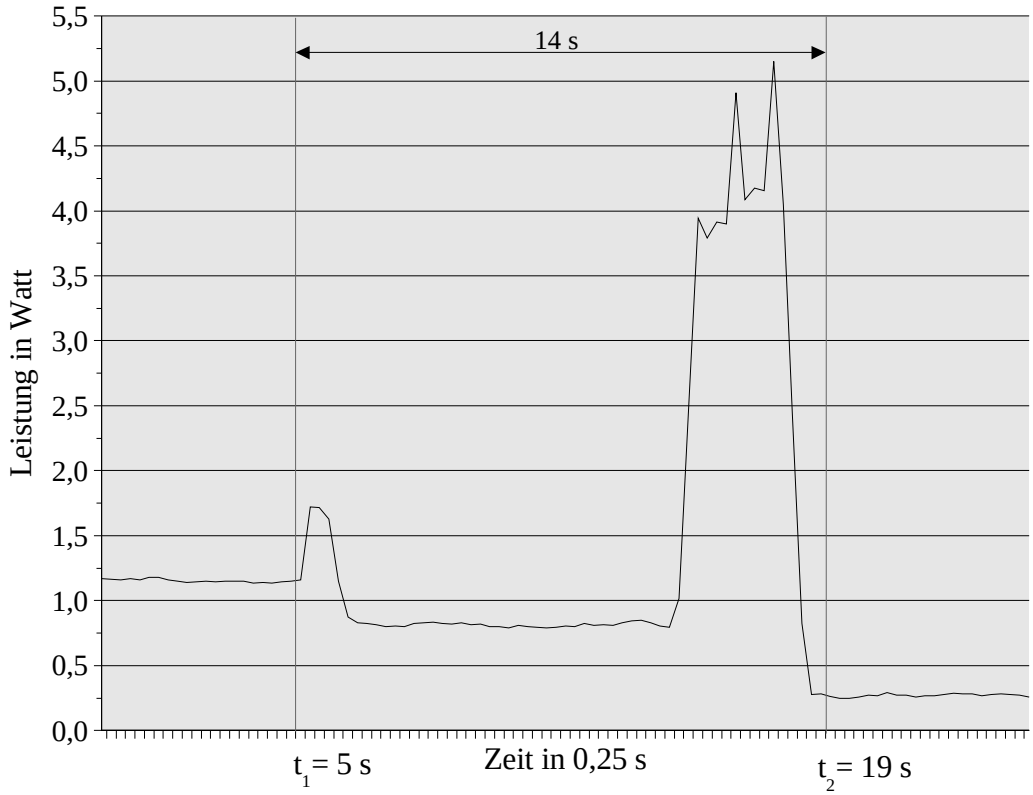

Abbildung 3: Leistungsaufnahme beim Wechsel von Low Performance Idle nach Standby

Der Wechsel in den Standby Mode ist dabei zum Zeitpunkt t_1 = 5 s durch Absetzen eines Standby−Immediate Kommandos ausgelöst worden. Direkt im Anschluss scheint die Platte im Low Performance Idle Mode bereits abgeschaltete, aber für die Powerdown−Sequenz benötigte Elektronik nochmal kurzzeitig zu aktivieren, wodurch der temporäre Anstieg zu erklären ist. Jetzt wird deutlich hörbar der Spindel−Motor abgeschaltet, woraufhin sich die Leistungsaufnahme bei ungefähr 0,8 Watt einpendelt. Dieser Wert bleibt für einige Sekunden konstant, bis laut Spezifikation der Head−Lock Mechanismus startet, der die Köpfe in Parkposition sichert. Dieser Vorgang ist ebenfalls hörbar und mit einer großen Leistungsaufnahme verbunden.

Energetisch gesehen ist der gesamte Vorgang also erst nach etwa 14 Sekunden abgeschlossen.

Wie sich aus Abbildung 3 bereits erahnen läßt, ist der Wechselvorgang mit einem hohen Aufwand an Energie verbunden. Dieser soll nun aus den Messwerten nummerisch ermittelt werden.

Energieverbrauch Fall A: Wechsel von Low Performance Idle nach Standby

Nach [BKL99] ist die verbrauchte Energie E bei aufgenommener Leistung P(t) im Intervall t_1 bis t_2 definiert als:

$$E = \int_{t_1}^{t_2} P(t)\,dt$$

Da es sich in diesem Fall um diskrete Messwerte handelt (n Werte bei einer Rate von 4 s^{-1}) vereinfacht sich die Formel auf:

$$E = \frac{1}{4}\sum_{1}^{n} P_i$$

Wendet man diese Formel auf die Messwerte aus Abbildung 3 im Intervall t_1 bis t_2 an, so ergibt sich ein Wert von 21,76 Joule.

Fall B: Wechsel von Standby nach Active

Es wurde, analog zu Fall A, eine Messreihe von 10 Einzelmessungen angelegt. Der Wechsel erfolgte jeweils 5 Sekunden nach Beginn der Messung durch einen einzelnen write−Systemaufruf (gefolgt von sync(), um einen Plattenzugriff zu forcieren). Dabei ist die Kurve aus Abbildung 4 entstanden.

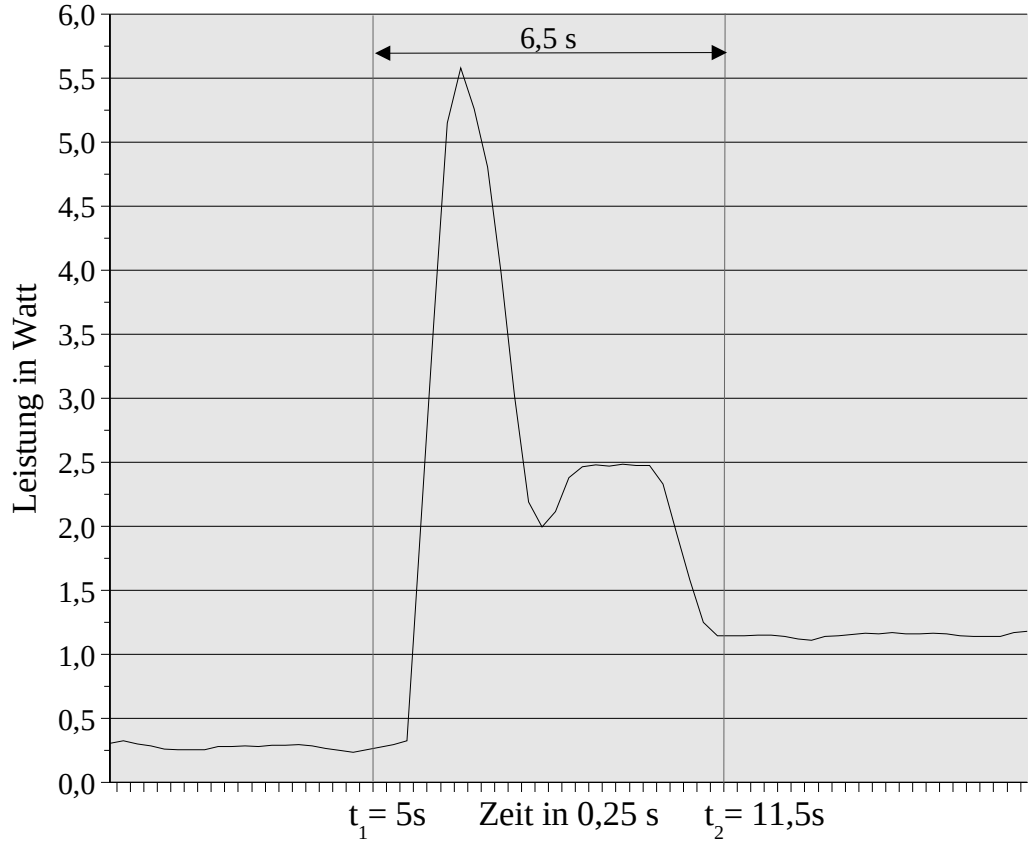

Abbildung 4: Leistungsaufnahme beim Wechsel von Standby nach Active (1 write−Aufruf)

Zum Zeitpunkt t_1 wird ein write−Systemaufruf getätigt. Der darauf folgende, steile Anstieg der Leistungsaufnahme ist auf das Anlaufen des Spindel−Motors zurückzuführen. Was allerdings danach passiert, ist aus dieser Messkurve nicht auf den ersten Blick zu ersehen. Es stellt sich für knapp 2 Sekunden ein Wert von etwas weniger als 2,5 Watt ein und schließlich sinkt die Leis−tungsaufnahme wieder auf den Wert von Low Performance Idle Mode ab. So liegt die Vermutung nahe, dass der einzelne write−Systemaufruf zu schnell abgearbeitet worden ist und damit die Platte zu kurz im Active Mode gewesen ist, als dass sich dies bei einer Aufzeichnung von 4 Werten pro Sekunde in der Messkurve niederschlagen könnte. Das Einpendeln bei 2,5 Watt könnte damit auf ein kurzes Verweilen im Performance Idle Mode zurückzuführen sein.

Um diese Annahme zu überprüfen, ist eine zweite Messreihe notwendig. Bei sonst unveränderten Parametern sind jetzt jeweils 40 write−sync Aufrufe abgesetzt worden.

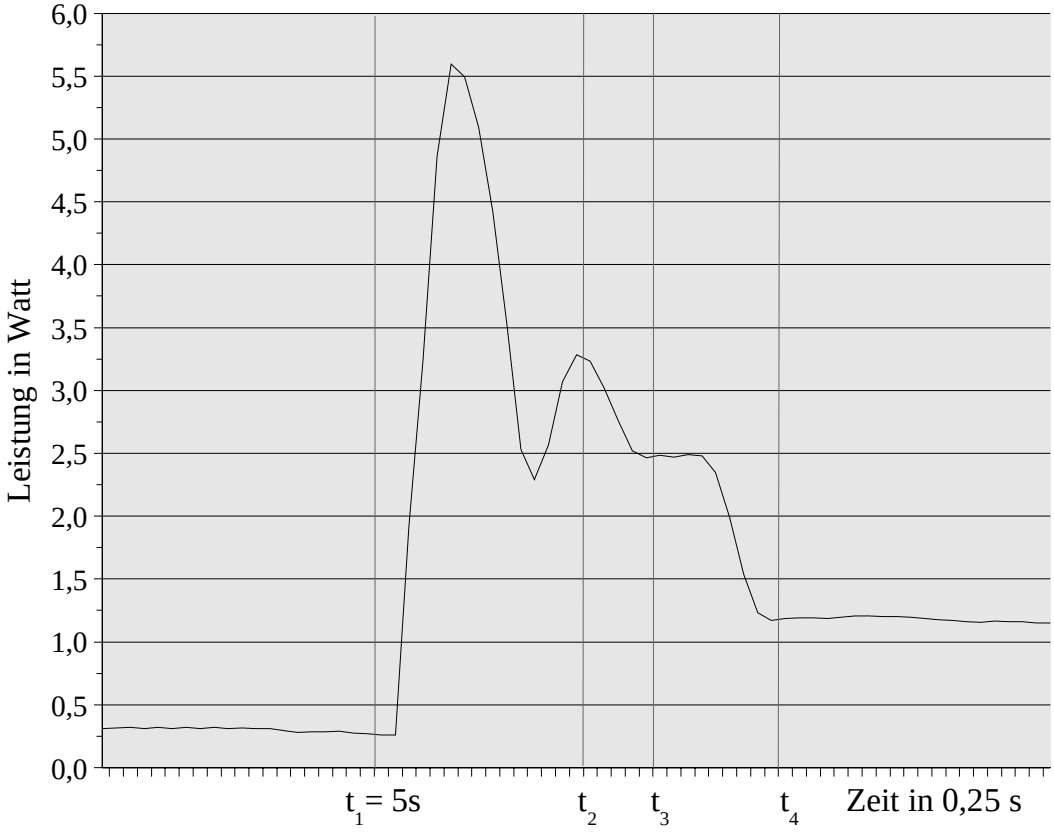

Abbildung 5: Leistungsaufnahme beim Wechsel von Standby nach Active (40 write−Aufrufe)

Wieder steigt die Leistungsaufnahme, bedingt durch das Anlaufen der Platte, zuerst stark an. Allerdings pendeln sich die Werte jetzt nicht sofort bei 2,5 Watt ein, sondern steigen nochmals für eine Sekunde auf ungefähr 3,3 Watt (bei t_2). Dabei handelt es sich um den Wert im Active Mode, die 40 write−Systemaufrufe halten die Platte jetzt lange genug aktiv. Anschliessend fällt der Wert wieder auf die schon in Abbildung 4 ersichtliche Leistungsaufnahme des Performance Idle Mode von knapp 2,5 Watt ab (t_3). Nach einer weiteren Sekunde schließlich wird in den Low Performance Idle Mode gewechselt (t_4).

Energieverbrauch Fall B: Wechsel von Standby nach Active

Wie bereits erwähnt, wird der Wechsel von Active nach Performance Idle und weiter nach Low Performance Idle bei dieser Betrachtung mit berücksichtigt. Da sich der in Abbildung 5 gezeigte Effekt erst bei mehr als 10 write−Aufrufen bemerkbar macht (siehe dazu Messungen im Fall C), wurden die Werte aus Abbildung 4 als Berechnungsgrundlage gewählt.

Nach der oben bereits verwendeten Formel ergibt sich für den Zeitraum t_1 bis t_2 ein Energie−verbrauch von 16,29 Joule.

Fall C: Wechsel von Low Performance Idle nach Active

Eine weitere Messreihe beschreibt die Leistungsaufnahme in diesem Fall. Wie aus den Abbildungen 4 und 5 bereits zu vermuten ist, steigt die Leistung zuerst steil auf den Wert des

Active Mode (3,29 Watt) an und fällt sofort nach Bearbeitung des letzten Lese−, Schreib− oder Positionierungskommandos auf den Wert des Performance Idle Mode (2,49 Watt) ab. Nach einer leicht variierenden Zeitdauer (wie bereits angeführt, wird sie von einem platteneigenen Algorithmus bestimmt) wechselt die Platte in den Low Performance Idle Mode, der Wert sinkt auf 1,15 Watt.

Da der Energieverbrauch in diesem Fall stark davon abhängt, wie viele Lese−, Schreib− oder Positionierungskommandos kurz hintereinander ausgeführt werden (solange sich die Platte noch im Performance Idle Mode befindet), können hier keine allgemein gültigen Aussagen gemacht werden. Abbildung 6 zeigt daher die Leistungsaufnahme bei 1, 10, 40 und 80 write−Systemaufrufen.

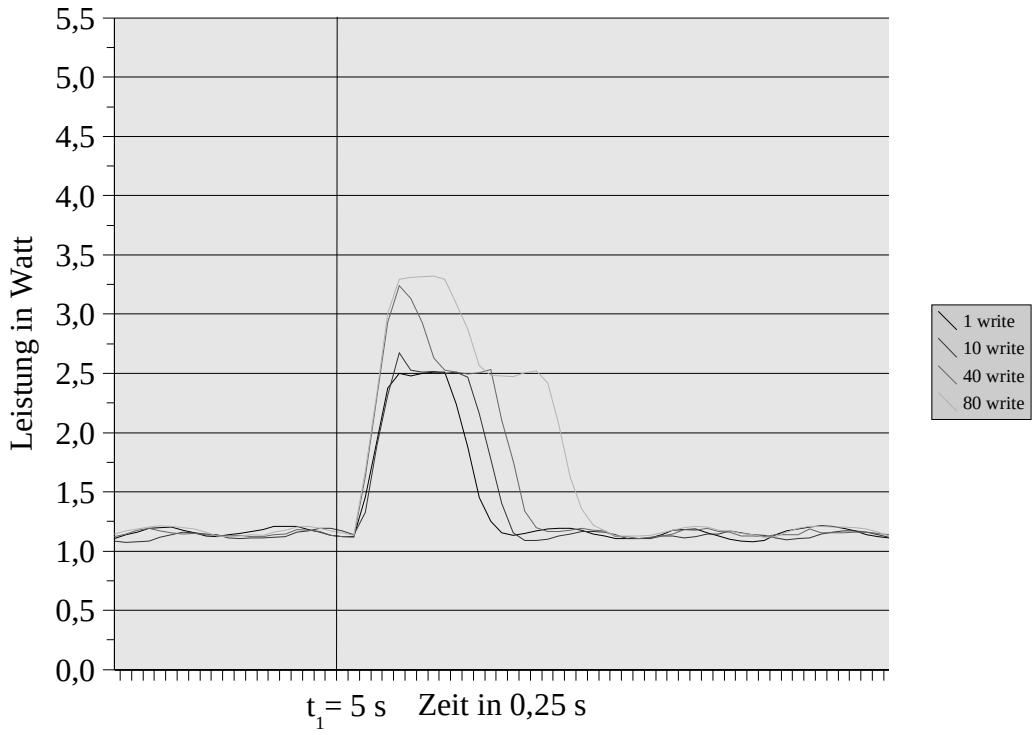

Abbildung 6: Leistungsaufnahme beim Wechsel von Low Performance Idle nach Active

Wie in den vorherigen Messungen ist auch hier 5 Sekunden nach Beginn der Messung der erste write−Aufruf abgesetzt worden. Die Leistungsaufnahme steigt stark an, da die Platte in den Active Mode wechselt. Das Energieniveau des Active Mode (durchschnittlich 3,29 Watt) errei−chen nur die Graphen bei 40 und 80 write−Aufrufen, da sonst die zeitliche Messauflösung von 0,25 s nicht ausreicht, um den kurzzeitigen Anstieg aufzuzeichnen. In jedem Fall wird für eine Dauer von 1,25 bis 1,75 Sekunden im Performance Idle Mode verblieben, bis wieder in den Low Performance Idle Mode gewechselt wird.

Energieverbrauch Fall C: Wechsel von Low Performance Idle nach Active

Im Gegensatz zu den Fällen A und B kann hier kein fester Wert berechnet werden, da zu viele unbekannte Größen existieren. So ist, um den zeitlichen Rahmen dieser Arbeit nicht zu sprengen, der Zusammenhang zwischen der Anzahl der write−Aufrufe und der Zeit, die infolge dessen im Active Mode verbracht wird, nicht intensiver untersucht worden. Allerdings würden diese

Ergebnisse in der Praxis keinen echten Schritt vorwärts bedeuten. Denn weitere Messungen haben ergeben, dass die Zeit, die im Performance Idle Mode verbracht wird um bis zu einer Sekunde schwankt. Wie erwähnt, wird diese Zeit von der Platte selbst bestimmt und kann nicht vom Hostsystem beeinflusst werden. Darüber hinaus hat das Hostsystem keine Möglichkeit festzustellen, ob sich die Platte im Performance Idle Mode oder Low Performance Idle Mode befindet. Aus diesem Grund kann der Energieverbrauch für diesen Fall nicht genau bestimmt werden.

Die in Abbildung 6 gemessenen Kurven treten in der Praxis nicht immer in dieser unverfälschten Form auf. Trifft im Performance Idle Mode bereits eine weitere Anforderung ein, so wird unmittelbar in den Active Mode gewechselt. Dieser Fall ist für das Hostsystem nicht fassbar, da es mangels Abfragemöglichkeiten auf Schätzungen bezüglich des aktuellen Energiemodus (Performance Idle oder Low Performance Idle Mode) angewiesen ist.

Auffällig ist aber, dass es für die Energieaufnahme offensichtlich wenig relevant ist, wie viele write−Aufrufe getätigt werden, solange sie direkt aufeinander folgen und nicht viel mehr als 10 sind. Die Graphen für 1 und 10 Aufrufe in Abbildung 6 unterscheiden sich so geringfügig, dass ein nicht zu unterschätzendes Einsparpotential darin liegt, Aufrufe zeitlich zu synchronisieren. Numerische Analysen haben ergeben, dass der Energieverbrauch bei 10 write−Aufrufen im Durchschnitt nur etwa 10% über dem eines einzigen liegt. Der Grund dafür ist in der Tatsache zu suchen, dass der große Teil der Kosten nicht durch den Festplattenzugriff selbst verursacht wird, sondern durch den anschließenden Verbleib im wesentlich teureren Performance Idle Mode für etwa 1,5 Sekunden.

2.2.3. Folgerungen für ein Energiesparkonzept

Im vorausgegangenen Kapitel wurde festgestellt, dass nur ein Wechsel vom Low Performance Idle Mode in den Standby Mode, sowie vom Standby Mode in den Active Mode sinnvoll vom Hostsystem kontrolliert werden kann und somit energetisch kalkulierbar ist. Daher muss sich ein betriebssystemeigenes Energiesparkonzept darauf beschränken, die Platte unter gewissen Bedingungen in den Standby Mode zu versetzen und unter anderen Bedingungen wieder zu reaktivieren.

Dabei können die obigen Messwerte gewisse Rahmenbedingungen schaffen. Wie aus den Abbildungen 3, 4 und 5 leicht abzulesen ist, ist sowohl der Wechsel in den Standby Mode als auch aus dem Standby Mode heraus ein energetisch kostspieliger Prozess. Daraus folgt, dass es eine Mindestzeit gibt, die im Standby Mode verblieben werden muss, damit sich überhaupt ein Einspareffekt ergibt.

Berechnung der minimalen Standby Zeit

Die Standby Zeit t_{SB} setzt sich aus 3 Komponenten zusammen, wie Abbildung 7 zeigt.

Abbildung 7: Komponenten der minimalen Standby Zeit

Powerdown Zeit t_{PD} und Spinup Zeit t_{SU} sind dabei in ihrer Länge fest, wie die Messungen im letzten Kapitel gezeigt haben. Die effektive Standby Zeit t_{SBeff} dagegen ist variabel und damit die Größe, die es zu berechnen gilt. Formell gilt offensichtlich folgender Zusammenhang:

$$t_{SB} = t_{PD} + t_{SBeff} + t_{SU}$$

Soll durch das Herunterfahren der Platte ein Einspareffekt erzielt werden, so muss während der Zeit weniger Energie verbraucht werden, als die Platte in der gleichen Zeitspanne im Low Performance Idle Mode aufgenommenen hätte. Die Energie E_{SB} , die verbraucht wird, wenn die Platte für die Zeitspanne t_{SB} heruntergefahren wird, berechnet sich als

$$E_{SB} = E_{PD} + P_{SB} \cdot t_{SBeff} + E_{SU}$$

wobei E_{PD} die im letzten Kapitel berechnete Energie während der Powerdown Zeit bezeichnet, E_{SU} die entsprechende Energie während der Spinup Zeit (ebenfalls bereits berechnet). P_{SB} ist die Leistungsaufnahme im Standby Mode.

Die Energie E_{LPI} , die während der gleichen Zeitspanne t_{SB} im Low Performance Idle Mode aufgenommenen wird, berechnet sich als

$$E_{LPI} = P_{LPI} \cdot t_{SB}$$

wobei P_{LPI} die Leistung im Low Performance Idle Mode bezeichnet. Da gelten soll $E_{SB} < E_{LPI}$ berechnet sich t_{SBeff} als

$$t_{SBeff} > \frac{E_{PD} + E_{SU} - P_{LPI}(t_{PD} + t_{SU})}{P_{LPI} - P_{SB}}$$

Mit den schon bekannten Werten

E_{PD}	21,76 Joule
E_{SU}	16,29 Joule
t_{PD}	14 Sekunden
t_{SU}	6,5 Sekunden
P_{LPI}	1,15 Watt
P_{SB}	0,27 Watt

ergibt sich (gerundet) $t_{SBeff} > 16{,}5\,s$ beziehungsweise $t_{SB} > 37\,s$.

Für das Hostsystem ist es von größerer Bedeutung, wieviel Zeit zwischen Powerdown− und Spinup−Kommando mindestens liegen muss. Diese Zeit wird berechnet durch $t_{SB} - t_{SU}$ und liegt bei 30,5 Sekunden.

Kosten beim vorzeitigen Verlassen des Standby Mode

Muss aus einem bestimmten Grund (z.B. dringender Festplattenzugriff) der Standby Mode früher verlassen werden, so ist durch das Wechseln der Energiemodi mehr Energie verbraucht worden, als eingespart werden konnte. Um diesen Fall in einem Energiesparkonzept richtig bewerten zu können, ist es wichtig, die Höhe des Mehraufwands zu kennen.

Aus den eben bereits hergeleiteten Formeln berechnen sich diese Kosten einfach durch

$$C = E_{SB} - E_{LPI}$$

beziehungsweise

$$C(t) = [E_{PD} + P_{SB}(t - t_{PD}) + E_{SU}] - [P_{LPI}(t + t_{SU})] \qquad t \in (t_{PD}; \infty)$$

Abbildung 8: Energiemehraufwand beim vorzeitigen Verlassen des Standby Mode

wobei t die Zeit bezeichnet, die seit dem Powerdown−Kommando vergangen ist. Die Formel gilt nur, wenn $t \geq t_{PD}$, da ein Spinup während der Powerdown−Sequenz nur unter großem analytischen Aufwand (wenn überhaupt) in einer Formel zu fassen ist. Für Werte größer als 30,5 Sekunden liefert die Formel negative Werte und damit die Energieersparnis durch den Wechsel. Abbildung 9 zeigt die Auswirkung eines vorzeitigen Verlassens des Standby Modes graphisch.

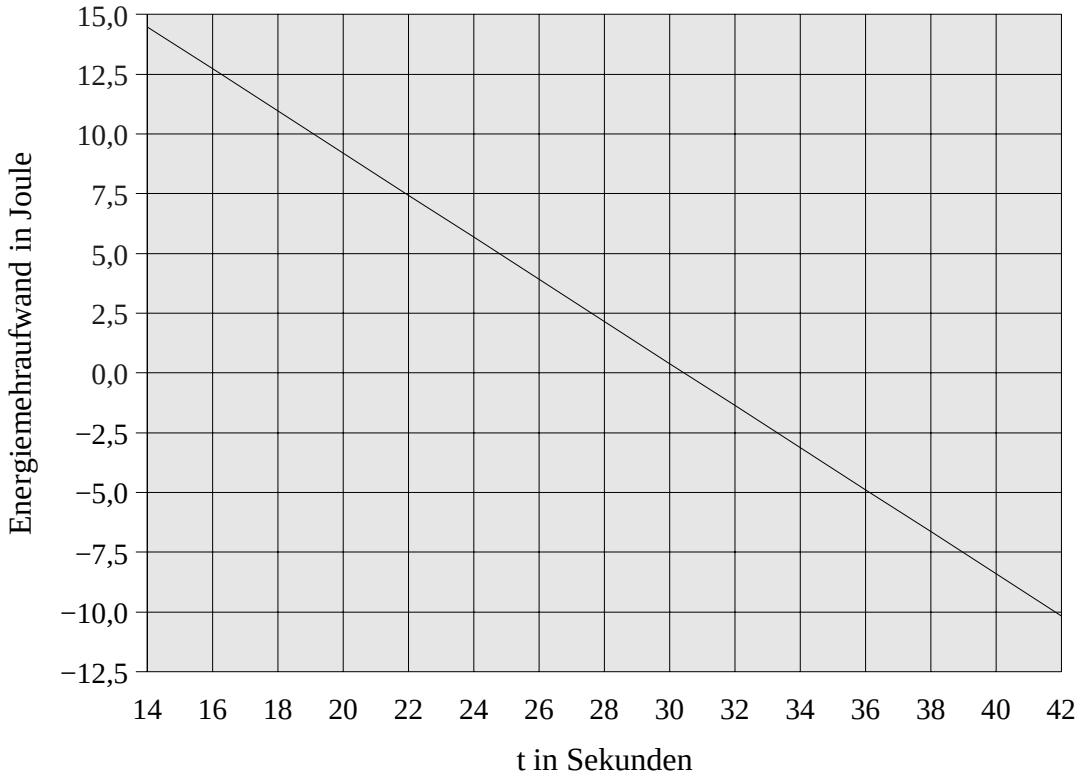

Abbildung 9: Energiemehraufwand bei vorzeitigem Verlassen des Standby Mode

Zusammenfassung

1. Ein Energiesparkonzept sollte sich darauf beschränken, Entscheidungen über das Betreten und Verlassen des Standby Modes zu fällen. Alle anderen Sparmodi können nur unzureichend vom Hostsystem kontrolliert werden.
2. Obwohl keine exakten Untersuchungen vorliegen, ist damit zu rechnen, dass das zeitliche Synchronisieren von Festplattenzugriffen ein nicht unbedeutendes Einsparpotential birgt (Fall C). Bis zu 10 direkt aufeinander folgende Zugriffe können energetisch (fast) wie ein einziger betrachtet werden.
3. Wird in den Standby Mode gewechselt, so sollte frühestens 30,5 Sekunden später ein Kommando zum Verlassen dieses Modes abgesetzt werden. Anderenfalls ist es energetisch günstiger, die Platte durchgängig im Low Performance Idle Mode zu betreiben.
4. Es ist möglich, Energieersparnis oder Mehraufwand in Abhängigkeit von der Zeit, die im Standby Mode verblieben wird, durch eine Formel zu berechnen. Dieser Wert kann bei der Entscheidungsfindung als aussagekräftiges Kriterium verwendet werden.

3. Konzept der kooperativen Nutzung

3.1. Grundidee

In dieser Arbeit soll ein Ansatz verfolgt werden, bei dem die Energieeinsparung durch kooperative Nutzung der Festplatte erzielt wird. Die Grundidee dabei ist, einen Marktplatz zu schaffen, wobei Prozesse Energie anbieten, um Leistungen vom Betriebssystem zu erhalten. Jeder Prozess verfügt über ein Energiekonto, von dem ausgegebene (d.h. von ihm verbrauchte) Energie abgezogen wird. Will ein Prozess eine Aktion tätigen, die Energie benötigt, so bietet er dem Betriebssystem einen gewissen Energiewert an. Das Betriebssystem entscheidet, ob es wirtschaftlicher ist, die Aktion sofort auszuführen, oder diese zu verzögern, bis andere Prozesse ebenfalls Aktionen ausführen wollen. Auf den Begriff der Wirtschaftlichkeit wird gleich noch genauer eingegangen. Auf diese Weise können sich die Gebote mehrerer Prozesse ansammeln, so dass in der Summe genug Energie zusammenkommen kann, um die geforderten Aktionen wirtschaftlich ausführen zu können.

Um diesen Ansatz zu konkretisieren, sollen die anfallenden Aufgaben auf zwei Komponenten verteilt werden. Der Marktplatz ist eine dieser Komponenten. Prozesse geben hier ihre Gebote ab, Preise werden ausgehandelt. Gegebenenfalls verfällt auch ein Gebot und wird durch ein höheres erneuert. Das Entgegennehmen und Verwalten der Gebote ist eine der Hauptaufgaben des Marktplatzes. Desweiteren ist eine Komponente notwendig, die das Betriebssystem auf dem Marktplatz vertritt. Es ist notwendig, zu entscheiden, wieviel eine Aktion kostet. Dies hängt eng mit dem Modus zusammen, in dem sich die Festplatte befindet. Da diese Komponente neben der Preisfindung vor allem für das Verwalten und Wechseln der Festplattenmodi zuständig ist, wird sie Modusverwalter genannt.

Daneben muss ein System geschaffen werden, mit dem Energie bemessen und verwaltet werden kann. Dafür ist die Einführung von Konten sinnvoll. Auf diese Weise kann der Energieverbrauch leicht reglementiert werden, damit kein Prozess in der Lage ist, mehr Energie zu bieten, als ihm zusteht.

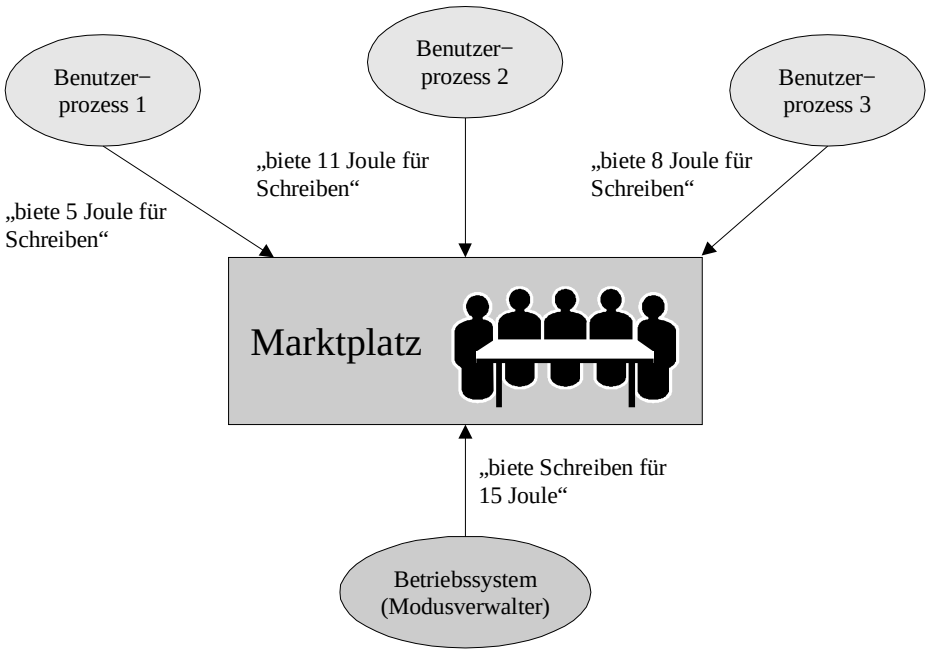

Abbildung 10: Auf dem Marktplatz treffen sich Angebot und Nachfrage

3.2. Marktplatz

Der Marktplatz ist die zentrale Komponente, hier treffen Angebot und Nachfrage aufeinander. Prozesse geben ihre Gebote dafür ab, wieviel Energie sie für welche Aktion bereit sind zu zahlen. Das Betriebssystem (der Modusverwalter) legt die Preise für die Aktionen fest. Es ist der einzige Anbieter und kalkuliert die Preise nach dem momentanen Energieaufwand, der unter Einbeziehung der Überlegungen aus dem letzten Kapitel nötig ist. Dies bedeutet, dass nicht der Preis für eine bestimmte Leistung ausgehandelt wird, sondern ein Wettbewerb der Benutzerprozesse um den Anteil entsteht, den jeder zu zahlen hat. Prinzipiell können Aktionen nur ausgeführt werden, wenn eine Zahl von Prozessen die notwendige Energie in Form von Geboten aufbringt.

Dies bedeutet, dass Prozesse auf dem Marktplatz gegebenenfalls auf Mitinteressenten warten müssen. Um zu verhindern, dass Prozesse wegen eines zu geringen Gebots große Verzögerungen hinnehmen müssen, verfällt das Gebot nach einer vom Prozess vorgegebenen Zeit. Dem Prozess bleibt die Möglichkeit ein neues, höheres Gebot zu platzieren oder anderweitig weiterzuarbeiten. Die Gebote des Betriebssystems haben eine kurze Verfallszeit und werden periodisch an die aktuelle Situation angepasst.

Haben sich schließlich genug Interessenten gefunden, ist der Handel perfekt und die gebotene Energie wird von den Benutzerprozessen an den Marktplatz als Instanz gezahlt. Dieser transferiert sie dann zum Modusverwalter. Der Marktplatz hat somit die Rolle eines Agenten. Er vermittelt dem Modusverwalter Kunden, indem er die Gebote einzelner Prozesse sammelt, bis der geforderte Preis gezahlt werden kann. Anschließend kassiert er die Energie bei den beteiligten Prozessen und übermittelt sie an den Modusverwalter.

3.3. Energiekonten

Prozesse können auf dem Marktplatz nicht beliebig viel Energie bieten. Schließlich ist die Energie in mobilen Systemen ein begrenztes Gut, und durch die Akkukapazität begrenzt. Um nicht jedem Prozess uneingeschränkten Zugriff auf vorhandene Energie zu geben, ist die Einführung von prozessspezifischen Konten („Prozesskonten“) sinnvoll. Dazu wird der Prozesskontext um einen Energiewert erweitert. Jeder Prozess bekommt bei seiner Entstehung einen gewissen Anteil vom Konto seines Vaterprozesses auf sein eigenes Konto überschrieben. Damit muss er nun seine Aufgaben bestreiten. Verbrauchte Energie wird vom Konto abgezogen. Ist es leer, kann er keine Energie mehr anbieten und ist damit darauf angewiesen, zu warten, bis sich genügend andere Interessenten finden, die sein Vorhaben mittragen. Stirbt ein Prozess, so erbt sein Vater die verbliebene Restenergie.

Beim Start des Systems wird dem Init−Prozess als erstem Benutzerprozess ein vom Administrator vorkonfigurierter Energiewert zugewiesen. Dies ist der einzige Vorgang, bei dem formell Energie „entsteht“. Da alle weiteren Prozesse durch fork−Aufrufe erzeugt werden, verteilt sich die Energie im laufenden System. Solange keine Zugriffe auf die Festplatte erfolgen, bleibt die Summe der Energie in den Prozesskonten konstant. Erst wenn auf dem Marktplatz eine Einigung erzielt ist, wird den beteiligten Prozessen Energie entsprechend ihrer Gebote abgezogen. Die so gesammelte Energie wird dem Modusverwalter „überwiesen“, der dafür ein eigenes Konto („Modusverwalterkonto“) hat. Nur der Modusverwalter darf Energie „vernichten“ (d.h. für verbraucht erklären). Er tut dies in dem Maße, in dem tatsächlich Energie verbraucht wird. Die tatsächlich verbrauchte Energie wird theoretisch ermittelt. Die Basis hierfür bilden die Untersuchungen aus Kapitel 2.2. Alle anderen Vorgänge erhalten die Energie, es handelt sich lediglich um Umbuchungen auf andere Konten.

Analysiert man den Energieverbrauch einer Festplatte, so sind zwei Anteile zu unterscheiden:

1. ein konstanter Anteil, der auch anfällt, wenn nicht auf die Platte zugegriffen wird („Basis−energieverbrauch“)
2. ein variabler Anteil, der bei Zugriffen zusätzlich verbraucht wird.

Die Energie in den Prozesskonten kann nur dazu verwendet werden, zusätzlichen Energiever−brauch aufgrund von Zugriffen abzurechnen. Der Basisenergieverbrauch der Festplatte wird nicht durch die Aktivität eines speziellen Prozesses ausgelöst, sondern fällt unabhängig davon ständig an. Um ihn einzubeziehen, existiert deshalb ein betriebssystemeigenes Konto („Be−triebssystemkonto“), das vom Modusverwalter administriert wird. Dem kontinuierlichen Basisenergieverbrauch wird dadurch Rechnung getragen, dass der Modusverwalter periodisch (jede Sekunde) Energie entsprechend dem Verbrauch des aktuellen Modus, in dem sich die Platte befindet, abzieht. Dieses Konto muss ebenfalls beim Systemstart vom Administrator mit einer Startenergie initialisiert werden.

Der Energieverbrauch der Festplatte wird also dadurch abgerechnet, dass der Basisenergiever−brauch vom Betriebssystem gezahlt wird (über das Betriebssystemkonto des Modusverwalters). Alle Kosten durch zusätzliche Zugriffe müssen die Verursacher, sprich die Benutzerprozesse, mit der Energie in ihren Prozesskonten bezahlen. Das Modusverwalterkonto dient dagegen nur als Übertragskonto, auf das nach erfolgter Einigung auf dem Marktplatz jeder Prozess seinen Anteil am Gesamtenergieaufwand für die geplante Aktion einzahlt.

3.4. Modusverwalter

Der Modusverwalter ist die Instanz innerhalb des Betriebssystems, die Entscheidungen über den Wechsel in einen anderen Energiemodus trifft. Darüber hinaus bestimmt er den Preis, der am Marktplatz für Aktionen gefordert wird und rechnet den Basisenergieverbrauch ab.

3.4.1. Abrechnung des Basisenergieverbrauchs

Der Basisenergieverbrauch (unabhängig von Aktionen) wird von Modusverwalter vom Betriebssystemkonto abgezogen. Dabei sind nur zwei Fälle zu unterscheiden:

1. die Platte befindet sich im Low Performance Idle Mode
2. die Platte befindet sich im Standby Mode

Entsprechend dem Energieverbrauch im aktuellen Modus wird vom Betriebssystemkonto jede Sekunde Energie abgezogen. Bei einem Wechsel in den jeweils anderen Modus wird die Abrechnung sofort nach dem Auslösen des Wechsels auf den neuen Modus umgestellt, so als wären die in Kapitel 2.2.2 ermittelten Verzögerungszeiten nicht existent. Die Differenz zwischen diesem abgerechneten Verbrauch und dem tatsächlichen sind Zusatzkosten des Wechsels und müssen anderweitig getragen werden (siehe dazu Kapitel 3.4.2).

Eine offene Frage ist, wie in den anderen Modi der Festplatte abgerechnet wird. Da der Sleep Mode nicht verwendet wird, bleibt nur Performance Idle Mode und Active Mode übrig. Die Verweildauer in diesen Modi ist sehr gering und außerdem kann das Hostsystem nicht feststellen, in welchem dieser Modi sich die Platte gerade befindet. Somit ist es sinnvoller, den Verbrauch im Low Performance Idle Mode weiterhin als Basisenergieverbrauch zu nehmen. Alle darüber hinaus gehenden Kosten sind dann als aktionsbezogen zu sehen und müssen damit über den Marktplatz bezahlt werden.

Ein Sonderfall tritt ein, wenn das Betriebssystemkonto leer ist. In diesem Fall muss der Basisenergieverbrauch auf die Kosten für Aktionen aufgeschlagen werden und von den Benutzerprozessen auf dem Marktplatz mitbezahlt werden.

3.4.2. Bestimmung der Preise für Aktionen

Die Preise müssen grundsätzlich den tatsächlichen Energieverbrauch decken. Dabei wird der Basisenergieverbrauch der Festplatte außer Acht gelassen. Grundsätzlich werden Lese- und Schreiboperationen gleich behandelt. Ein Problem ergibt sich, wenn Operationen kurz hintereinander ausgeführt werden. In diesem Fall können, wie in Kapitel 2.2.2 gezeigt, die Operationen nicht mehr einzeln gesehen werden, sondern müssen gemeinsam betrachtet werden. Da nur Low Performance Idle Mode und Standby Mode unterschieden werden, gibt es zwei Preiskategorien.

Preis im Low Performance Idle Mode

Der Preis ist fix. Er stellt einen Durchschnittswert der tatsächlichen Kosten für Aktionen in diesem Modus dar. Die Problematik aus Fall C in Kapitel 2.2.2 greift hier, so dass mit einer groben Näherung gearbeitet werden muss. Da, wie dort bereits bemerkt, mehrere Operationen, die kurz

hintereinander ausgeführt werden, energetisch kaum aufwendiger sind als eine einzelne, wird der Preis nicht pro Aktion erhoben. Stattdessen können sich mehrere Gebote zu diesem Preis auf−addieren. Da die bietenden Prozesse erst blockiert und anschließend deren Anforderungen direkt nacheinander abgearbeitet werden, ist dies zu rechtfertigen. Das Bieten auf dem Marktplatz sorgt in diesem Fall für eine Synchronisation der Aktionen und damit für eine weitere Energieeinspa−rung.

Preis im Standby Mode

Der Preis hängt von der Zeit ab, die bereits im Standby Mode verbracht wurde.

$$Preis_{Standby\ Mode}(t)=\begin{cases} \widehat{E_{SU}}+C(t_{PD}) & 0\le t<t_{PD} \\ \widehat{E_{SU}}+C(t) & t_{PD}\le t<t_{SB}-t_{SU} \\ \widehat{E_{SU}} & t_{SB}-t_{SU}\le t<\infty \end{cases}$$

Abbildung 11: Preis im Standby Mode

Die Bedeutung der Variablen wurde bereits in Kapitel 2.2.3 erklärt. Prinzipiell fallen stets die Kosten für den Spinup an. Da jedoch der Basisenergieverbrauch bereits vom Betriebssystem gezahlt wird, muss dieser Anteil von den Kosten abgezogen werden. Es gilt offensichtlich:

$$\widehat{E_{SU}}=E_{SU}-t_{SU}\cdot P_{LPI}$$

Die Kosten des Powerdown werden vom Betriebssystemkonto abgebucht und müssen damit beim hier zu ermittelnden Preis nicht berücksichtigt werden. Sollten jedoch nach der Formel aus Abbildung 8 zusätzliche Kosten aufgrund eines verfrühten Spinups anfallen, so erhöhen diese den Preis entsprechend (mittlere Zeile in der obigen Formel). Da der Term C(t) für Zeitwerte kleiner der Powerdown Zeit nicht gilt, muss für diesen Bereich eine gesonderte Regelung getroffen werden. Hier wird zur Vereinfachung stets so verfahren, als ob der Powerdown abgeschlossen wäre. Der Preis ist in diesem Fall fest und besteht aus der Summe der Spinupkosten und der zusätzlichen Kosten direkt nach dem Powerdown (obere Zeile). Ist genug Zeit vergangen, so hat sich der Powerdown energetisch ausgezahlt und der Term C(t) liefert negative Werte (Energieersparnis). Diese sollen jedoch nicht zu einer Preissenkung führen, weshalb hier dennoch der Preis für den Spinup anfällt (untere Zeile).

Die Idee hinter dieser Preispolitik ist einfach. Energie soll eingespart werden. Dazu „investiert“ das Betriebssystem (der Modusverwalter), indem es die Kosten für einen Powerdown übernimmt, mit der Aussicht, dass sich während der Zeit im Standby Mode diese Investitionen durch Energieeinsparung beim Basisenergieverbrauch auszahlen. Wollen Benutzerprozesse Operationen ausführen, so müssen sie die anfallenden Kosten übernehmen, die sich aus den Spinupkosten selbst und gegebenenfalls anfallenden Zusatzkosten aufgrund eines zu frühen Spinups zusammensetzen. Von der Energieersparnis aufgrund einer langen Verweildauer im Standby Mode profitiert nur das Betriebssystem durch niedrigere Werte beim Basisenergieverbrauch. Würden die Preise in diesem Fall (untere Zeile der Formel) analog der mittleren Zeile durch den Term C(t) weiter gesenkt (und damit die Energieersparnis über den Preis weitergegeben), so bestünde kaum mehr ein Anreiz für die Prozesse, Energie zu sparen. Selbst niedrige Gebote könnten dann sofort zu einem Spinup führen. Eine lange Zeit im Standby Mode spart aber viel Energie. Deshalb ist es sinnvoll, den Preis trotz Energieersparnis bei den Spinupkosten zu lassen.

3.4.3. Entscheidungsfindung beim Wechsel des Energiemodus

Es wird, wie dargelegt, nur über zwei mögliche Wechsel entschieden: in den Standby Mode oder aus dem Standby Mode heraus. Der Wechsel aus dem Standby Mode heraus wird dann veranlasst, wenn der auf dem Marktplatz geforderte Preis dafür bezahlt worden ist. Der Wechsel in den Standby Mode erfolgt nach einem simplen Timeout („Standby Timeout“), der global konfigurierbar ist. Treffen nach dem letzten Zugriff auf die Festplatte in dieser Zeit keine weiteren Anforderungen ein, entscheidet sich der Modusverwalter für einen Powerdown. Dabei ist es irrelevant, ob auf dem Marktplatz weitere Gebote vorliegen. Entscheidend ist nur, ob Anforderungen, die den Marktplatz bereits passiert haben, beim Modusverwalter eintreffen.

3.5. Schwachstellen

Das vorgestellte Konzept basiert auf dem marktwirtschaftlichen Prinzip von Angebot und Nachfrage. Der freie Wettbewerb hat zum Ziel, einen Preis zu finden, der sowohl die Angebotsseite, als auch die Nachfrageseite zufrieden stellt. Dazu beginnen beide Seiten mit einem Startgebot und machen dann im Laufe der Verhandlungen Zugeständnisse. Somit nähern sich die Gebote an und treffen sich im Idealfall bei einem Mittelwert, den beide Seiten bereit sind zu akzeptieren.

Das Konzept der kooperativen Nutzung weicht in einem wichtigen Detail von diesem Prinzip an. Der Modusverwalter hat eine Monopolstellung und kann während der Verhandlungen keine Zugeständnisse machen. Er ist an den realen Energieverbrauch der Festplatte gebunden. Damit stehen nur die Benutzerprozesse untereinander im Wettbewerb. Eine Preisfindung, eigentlich Hauptziel des Prinzips von Angebot und Nachfrage, findet nicht statt. Es gibt daher Konstellationen, in denen das Konzept nicht greift. Sind nur sehr wenige Benutzerprozesse vorhanden, so ist die Wahrscheinlichkeit gering, dass diese zur selben Zeit Aktionen ausführen. Will nur ein Prozess lesen oder schreiben, so muss er die notwendige Energie selbst aufbringen. Der Marktplatz ist in diesem Fall sinnlos, er verursacht eine unnötige Verzögerung. Es wird keine Energie eingespart.

Das Konzept stellt keine Prognosen für die Zukunft auf. Die Charakteristik der Zugriffe wird nicht analysiert, periodische Zugriffe können daher nicht erkannt werden. Ist der Standby Timeout ungünstig gewählt, wird kurz vor dem nächsten Zugriff in den Standby Mode gewechselt. Zwar verhindert der hohe Preis, dass sofort ein Spinup erfolgt, jedoch könnte mehr Energie eingespart werden, wenn der Powerdown erst kurze Zeit später erfolgen würde. Ein Möglichkeit der Verbesserung wird in Kapitel 5 vorgestellt.

4. Einbettung in das Betriebssystem MINIX

Im praktischen Teil ist das vorgestellte Konzept in das Betriebssystem MINIX zu integrieren. Im Folgenden wird deshalb genauer auf die dabei entstehenden Probleme, sowie deren Lösung eingegangen.

4.1. Überblick über den Aufbau von MINIX

4.1.1. Überblick über das System

Bei MINIX („mini−UNIX“) handelt es sich um ein Betriebssystem, das von Andrew S. Tanenbaum, dem Autor von [Tan97], entwickelt wurde mit dem Ziel, als gut strukturiertes, leicht verständliches praktisches Beispiel in Vorlesungen über Betriebssysteme zu dienen. MINIX ist aus Benutzersicht UNIX−kompatibel, unterscheidet sich jedoch in den Interna. MINIX gliedert sich in vier Schichten wie Abbildung 12 zeigt.

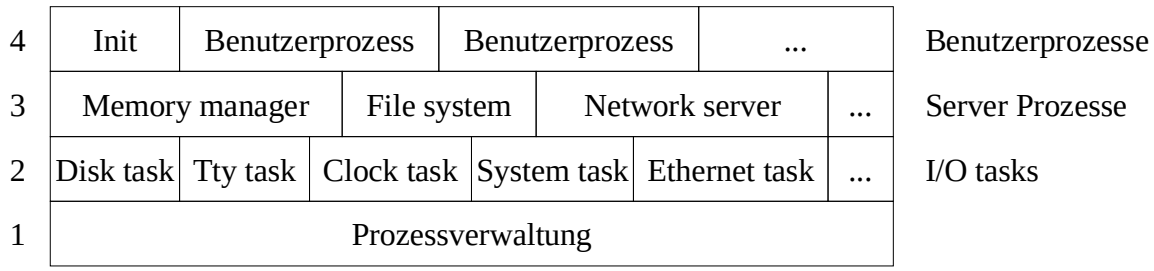

Abbildung 12: Schichten des MINIX−Systems

Die unterste Schicht bearbeitet Interrupts, betreibt Scheduling und stellt den höheren Schichten alles zur Verfügung, was notwendig ist, um das Prozess−Model lauffähig zu machen. Prozesse kommunizieren über Nachrichten. Dieser Mechanismus wird ebenfalls von Schicht 1 imple−mentiert.

In der zweiten Schicht sind alle I/O−Tasks zu finden (jeweils eine pro Gerätetyp). Der Begriff Task wird in [Tan97] verwendet, um eine Unterscheidung gegenüber Benutzerprozessen zu treffen. Sie sind die Gerätetreiber von MINIX. I/O−Tasks verwenden teilweise gemeinsame Unterprogramme, laufen jedoch ansonsten unabhängig und kommunizieren über Nachrichten. Die System Task stellt Dienste zur Verfügung, für die besondere Privilegien erforderlich sind, wie das Kopieren von Daten zwischen Adressräumen. Die Tasks aus Schicht 2 werden mit dem Code aus Schicht 1 zu einem großen Programm, dem Kern zusammengelinkt.

Prozesse der Schicht 3 stellen den Benutzerprozessen nützliche Dienste zur Verfügung. Systemaufrufe werden in dieser Schicht interpretiert. Der Memory Manager führt alle die Speicherverwaltung betreffenden Aufrufe aus (wie fork oder exec), das Filesystem, das als eine Art Fileserver implementiert ist, alle das Filesystem betreffenden (wie read, mount, chdir). Prozesse dieser Schicht unterscheiden sich kaum von Benutzerprozessen. Nur eine höhere

Priorität und die Tatsache, dass sie vor allen Benutzerprozessen gestartet und nie beendet werden, gibt ihnen einen kleinen Sonderstatus.

Schicht 4 schließlich enthält sämtliche Benutzerprozesse.

Auf eine Besonderheit von MINIX, die aus diesem Schichtenmodell resultiert, sei im Bezug auf den Prozesskontext (proc−Struktur) hingewiesen. Da mit Filesystem und Memory manager zwei Komponenten von zentraler Bedeutung aus dem Kern in eigene Adressräume ausgelagert sind, werden drei Teil−Kontexte verwaltet: Die kproc−Struktur (Kern−Prozesskontext), die mproc−Struktur (Memory Manager−Prozesskontext) und die fproc−Struktur (Filesystem−Prozesskontext). Die Indizierung erfolgt bei allen drei Strukturen einheitlich.

4.1.2. Überblick über den Ablauf von Festplatten−I/O

Im Folgenden soll das Zusammenspiel der Komponenten in den einzelnen Schichten am Beispiel eines read−Systemaufrufs illustriert werden. Im Falle, dass der zu lesende Block nicht im Cache des Filesystems gefunden wird, werden die in Abbildung 13 gezeigten Nachrichten versandt.

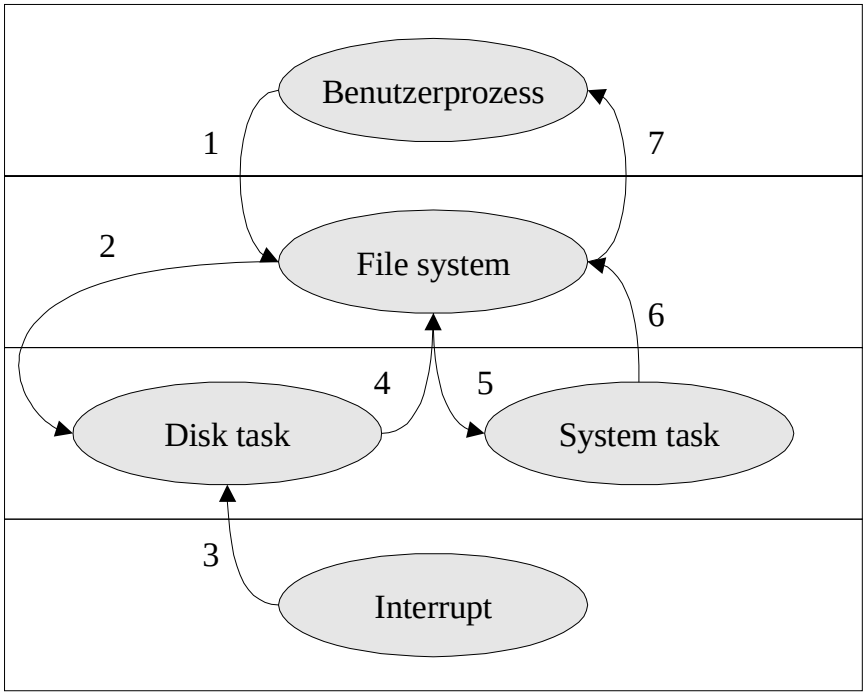

Abbildung 13 Ablauf beim Lesen von der Festplatte

1. Der Benutzerprozess tätigt einen read−Systemaufruf. Dadurch wird eine Nachricht an das Filesystem verschickt.
2. Das Filesystem findet den benötigten Block nicht im Cache und weist die Disk Task an, den Block zu laden.
3. Nach Abschluss des Ladens wird der eingegangene Interrupt von Schicht 1 in eine Nachricht an die Disk Task umgesetzt.
4. Die Disk Task liefert den geladenen Block an das Filesystem.

5. Das Filesystem weist die System Task an, die Daten in den Adressraum des Benutzerprozes−ses zu kopieren.
6. Die System Task bestätigt das erfolgreiche Kopieren.
7. Der read−Systemaufruf beim Benutzerprozess kehrt zurück.

4.2. Änderungen am Betriebssystem

4.2.1. Überblick

Um das in Kapitel 3 vorgestellte Konzept in MINIX zu integrieren, sind einige Änderungen notwendig. Die in Abbildung 12 gezeigten Komponenten in den einzelnen Schichten mussten angepasst und ergänzt werden. Es kommen die schon theoretisch behandelten Komponenten Marktplatz und Modusverwalter hinzu. Abbildung 14 zeigt das geänderte Schichtenmodell.

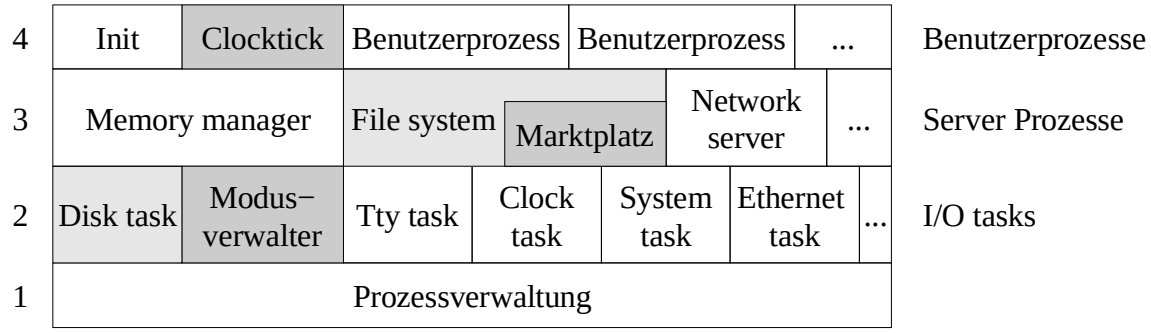

Abbildung 14: Angepasstes Schichtenmodell des MINIX−Systems

Die dunkelgrau hinterlegten Komponenten sind neu hinzugekommen, bei den hellgrau hinter−legten sind Änderungen vorgenommen worden. Die weißen Komponenten sind nicht oder nur unwesentlich modifiziert. Der Marktplatz ist nicht als eigener Prozess in Schicht 3 implemen−tiert, sondern in das Filesystem integriert. Der Grund dafür ist, dass der Marktplatz oft auf die Energiekonten der Prozesse zugreifen muss, die als Daten der Filesystem proc−Struktur nur im Filesystem bekannt sind. Eine Trennung von Marktplatz und Filesystem würde daher mit vielen zusätzlichen Nachrichten verbunden sein.

Der Modusverwalter ist eine eigene Schicht 2 Task. Eine eigene Task ist sinnvoll, da die Mög−lichkeit, in Energiesparmodi zu wechseln keine spezielle Eigenschaft von Festplatten ist. Sollen später andere Geräte in das Konzept aufgenommenen werden, ist dies mit relativ geringem Aufwand möglich. Die Ansiedlung in Schicht 2 (statt beispielsweise in Schicht 3) bringt dagegen keinen konzeptionellen Vorteil. Letztendlich ist der Aufwand zum Einbinden einer neuen Komponente in Schicht 2 jedoch kleiner.

Bei Clocktick handelt es sich um ein kleines Programm, das als Zeitsignalgeber für Timeoutbe−rechnungen benötigt wird. Um auf dem Marktplatz den Verfall eines Gebote nach einer gewissen Zeit zu realisieren und die verstrichene Zeit für den Standby−Timeout zu berechnen, müssen im Kern Zeitmessungen erfolgen. Auch die Preisberechnungen im Modusverwalter sind zeitabhän−gig. Da MINIX keine einfache Möglichkeit, beispielsweise über einen Timer, bietet, liefert der Clocktick−Prozess jede Sekunde ein Zeitsignal in Form einer speziellen Nachricht. Dies ist

notwendig, da der Empfang von Nachrichten grundsätzlich blockierend erfolgt. Bei einer andere Realisierung könnte das Filesystem/der Marktplatz daher nicht auf den abgelaufenen Timeout reagieren.

Um die Erläuterungen zu den einzelnen Komponenten besser verständlich zu machen, gibt Abbildung 15 einen Überblick über die Interaktionen. Pfeile deuten an, zwischen welchen Komponenten Nachrichten ausgetauscht werden.

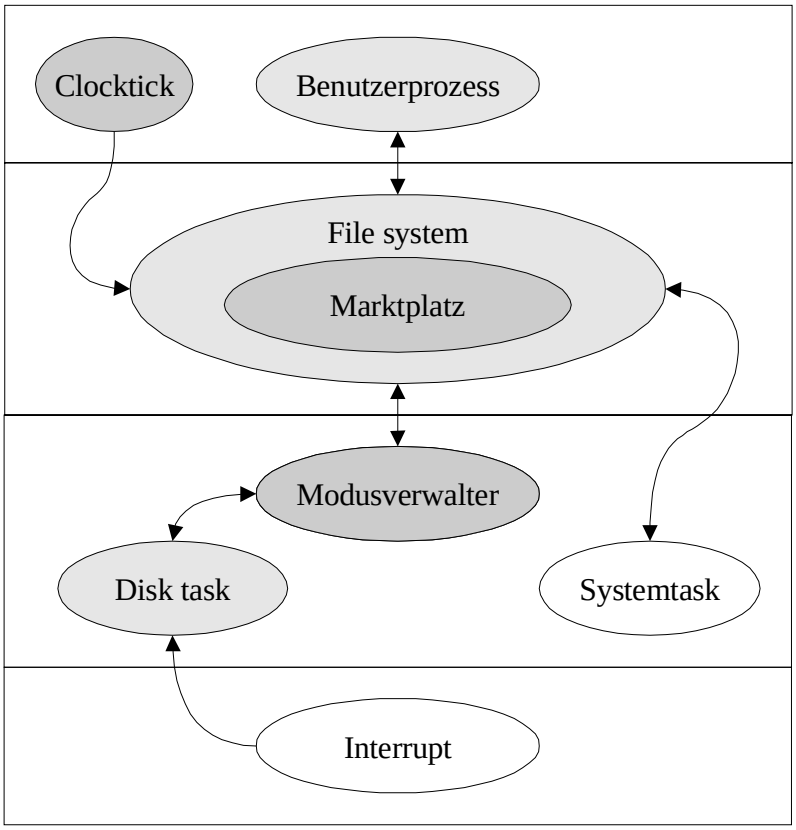

Abbildung 15: Kommunikationswege beim Festplattenzugriff

In den nachfolgenden Kapiteln soll auf die modifizierten und neu hinzugekommenen Komponenten im Detail eingegangen werden, bevor später das Zusammenspiel erläutert wird. Die Erklärung erfolgt von oben nach unten, beginnt also in Schicht 4 mit der Vorstellung der Änderungen aus Sicht der Benutzerprozesse.

Um das Konzept der kooperativen Nutzung in MINIX zu integrieren, ist eine Anzahl neuer Systemaufrufe nötig. Den Anwendungen soll die Möglichkeit gegeben werden, den Energieverbrauch ihrer Dateioperationen zu beschränken. Die neuen Systemaufrufe müssen in die C−Bibliothek eingebunden werden.

Allgemein wird ein Systemaufruf in MINIX dadurch realisiert, dass eine Nachricht an einen der Server−Prozesse aus Schicht 3 versendet wird. Ein Stub in der C−Bibliothek übernimmt dabei das Marshalling, also das Verpacken der Parameter in die Nachricht. Jedem Systemaufruf ist eine Kennziffer zugeordnet, verschiedene Varianten eines Aufrufs (wie dup, dup2) verwenden jedoch eine gemeinsame Kennziffer.

Um Kompatibilität mit vorhandenen Programmen zu gewährleisten, wurde für die Erweiterung von Systemaufrufen der gleiche Weg gewählt. Es existiert neben den erweiterten Aufrufen auch noch das Original−POSIX−Interface und beide Varianten teilen sich eine Kennziffer. Vorhandene Programme müssen allerdings neu übersetzt werden, da der dem POSIX−Interface entsprechende Stub zusätzlich Standardwerte für die erweiterten Parameter in die Nachrichten einträgt. Von Schicht 3 abwärts, ist also jeweils nur die neue Variante bekannt.

Diese Aufrufe teilen sich in zwei Gruppen:

1. Erweiterte Aufrufe
2. Neue Aufrufe

4.2.2. Erweiterung der POSIX−Aufrufe

Die hier vorgestellten, erweiterten Systemaufrufe unterscheiden sich von den ursprünglichen meist durch zwei Parameter: ein Energie−Gebot und dessen zugehörigen Timeout. Der Energiewert wird in 0,1 Joule, der Timeout−Wert in Sekunden angegeben.

Beide Werte sind vom Typ Integer, jedoch gelten spezielle Wertebereiche. Da zusätzliche Parameter in den Nachrichten, die vom Benutzerprozess an das Filesystem geschickt werden, untergebracht werden müssen, manche dieser Nachrichten aber nur noch ungenutzen Platz von der Größe eines 32−Bit−Integers enthalten, ist es notwendig beim Marshalling zwei Integer−Werte zu einem zu komprimieren. Durch entsprechende Arithmetik resultieren daraus die folgenden Wertebereiche:

energy:	−1	bis	262142	(18 bit)
timeout:	−1	bis	16382	(14 bit)

Das Packen und Entpacken wird von Makros übernommen. Bei einer Angabe von Werten außerhalb dieser Bereiche muss mit Überlauf−Effekten gerechnet werden.

Der Wert −1 nimmt bei beiden Parametern einen Sonderstatus ein. Ein Energiewert von −1 bedeutet „soviel wie verfügbar“. Der Wert entspricht dann dem aktuellen Stand des Energiekontos. Ein Timeout−Wert von −1 deaktiviert den Timeout. Das abgegebene Gebot verfällt dann nie. Das POSIX−Interface ohne Energieverwaltung entspricht dem erweiterten Interface mit Energiewert −1 und Timeout −1. Dadurch wird das Verhalten eines Systems ohne Energieverwaltung am besten nachempfunden. Dateioperationen werden möglichst umgehend ausgeführt und es treten keine Fehler aus „Energiemangel“ auf, die ein POSIX−Programm nicht kennt und daher nicht korrekt behandeln könnte.

Die Beschränkung des Energie−Gebots ist in der Praxis nicht von Bedeutung. Das maximal mögliche Energie−Gebot von 26214,2 Joule entspricht mehr als 15% der gesamten Akkukapazität in den meisten mobilen Systemen. Der Wert liegt im Bereich von einem Viertel der Akkukapazität des verwendeten Testsystems. Eine so große Energiemenge ist als Preis für eine einzelne Aktion unrealistisch. Ebenso bedeutet der maximale Timeout von über 4,5 Stunden praktisch keine Einschränkung, da die maximale Betriebsdauer im Akkubetrieb unter durchschnittlichen Bedingungen oft kürzer ist.

Es mag auf den ersten Blick verwundern, dass der Timeout nicht genauer als in 1−Sekunden−Schritten spezifiziert werden kann. Der Grund dafür liegt darin, dass MINIX keine Möglichkeit zur Verfügung stellt, Zeit genauer zu messen. Auf der anderen Seite sind sekundengenaue Messungen durchaus vertretbar, wenn man bedenkt, dass ein energetisch sinnvolle Powerdown−Spinup−Folge mehr als 37 Sekunden dauert.

Neben den zusätzlichen Parametern, erfordern die Erweiterungen aber auch eine angepasste Fehlerbehandlung. Da außer den üblichen Fehlern der Fall auftreten kann, dass ein Aufruf nach Ablauf des Timeouts aus Energiemangel unerledigt abgebrochen wird, muss ein neuer Fehlerwert eingeführt werden: `ENOENERGY`. Tritt der Fehler auf, so wird dieser − wie jeder andere Fehler − durch den Rückgabewert des Aufrufs signalisiert. Der Fehler kann dann durch Abfrage der globalen Variable errno identifiziert werden. Auch strerror und perror kennen diesen Fehlerwert und liefern einen passenden Text.

Bei der Implementierung des Energiesparkonzepts ist schnell klargeworden, dass eine Detail−Erweiterung sinnvoll ist. Anwendungsprogrammierer sehen Dateien als zusammengehörende Einheiten an. Ihnen ist wichtig, eine ganze Datei schnell und energiesparend zur schreiben. Da sehr viele Systemaufrufe mit Dateien arbeiten, ist es zudem sehr aufwendig, für jeden dieser Aufrufe ein neues Interface unter Berücksichtigung von Energie−Gebot und Timeout zu schaffen. Als Abhilfe bietet sich die Einführung von Energiekonten, die an Dateideskriptoren gebundenen sind („Dateikonten“) an.

Die Idee ist einfach. Ein open−Aufruf öffnet die Datei und überträgt einen gewissen Betrag vom Prozesskonto auf das Dateikonto. Alle Aufrufe, die diese Datei (diesen Dateideskriptor) betreffen, werden über das Dateikonto abgerechnet. Das bedeutet, dass gebotene Energie vom Dateikonto statt vom Prozesskonto abgezogen wird. Auf diese Weise lässt sich der Energieverbrauch für Operationen auf Dateiebene begrenzen, ohne dass aufwendiges Aufsummieren des Verbrauchs der einzelnen Systemaufrufe notwendig ist. Beim Schließen der Datei wird die Restenergie wieder dem Prozesskonto gutgeschrieben.

Anmerkung: Um die folgenden erweiterten Systemaufrufe nutzen zu können, müssen lediglich die Headerdateien der entsprechenden POSIX−Aufrufe eingezogen werden. Dort sind auch die erweiterten Aufrufe deklariert. Die Implementierung ist in die Standard−C−Bibliothek eingebunden.

Prozessbehandlung:

Syntax: `int fork2(int energy);`

Diese fork−Variante bietet die Möglichkeit, zu spezifizieren, wieviel Energie der Vaterprozess dem Kindprozess überschreibt. Die Energie wird vom Prozesskonto des Vaters abgezogen und dem Prozesskonto des erzeugten Kindprozesses gutgeschrieben. `energy` darf selbstverständlich nicht größer sein als der Wert des Vater−Prozesskontos. Die oben genannte Wertebereichseinschränkung gilt hier nicht.

Die POSIX−Variante überschreibt generell die Hälfte der Energie des Vaterprozesses auf den Kindprozess.

Syntax: `void exit(int status);`

Beendet den Prozess.
Der Energiewert des Prozesskontos wird dem Prozesskonto des Vaterprozesses gutgeschrieben.

Dateibehandlung:

Syntax:

```
int open2(   const char *path, int flags,
             int energy, int timeout[, mode_t mode]);
```

Öffnet eine Datei (Parameter *path*, *flags* und *mode* wie POSIX−Variante).

energy spezifiziert, wieviel Energie für Operationen auf dieser Datei maximal bereitgestellt werden soll. Der Wert hat zwei Bedeutungen. Einerseits spezifiziert er das Gebot, das für den open−Aufruf auf dem Marktplatz abgegeben werden soll. Andererseits ist die nach dem open verbleibende Energie der Startwert für das an den Dateideskriptor gebundene Konto (Datei−konto). Dabei bedeutet ein Wert von −1 „keine spezielle Beschränkung“ (alle auf dem Prozesskonto vorhandene Energie darf verwendet werden). *energy* muss kleiner oder gleich dem Wert des Prozesskontos sein.

timeout spezifiziert den Timeout in Sekunden, nach dem das Gebot für das Öffnen auf dem Marktplatz verfallen soll, falls dort auf Mitinteressenten gewartet werden muss. Ein Wert von −1 steht für unendlich und deaktiviert den Timeout.

Die POSIX−Variante entspricht den Parametern *energy = −1* und *timeout = −1*.

Syntax:

```
int close(int d);
```

Schließt eine Datei (exakt wie POSIX−Variante). Falls der Dateideskriptor d ein eigenes Konto hatte (siehe open2), wird es aufgelöst und die Restenergie dem Prozesskonto gutgeschrieben.

Syntax:

```
ssize_t read2(    int d, void *buf, size_t nbytes,
                  int energy, int timeout);
```

Versucht Daten von einer Datei zu lesen (Parameter *d*, *buf*, *nbytes* wie POSIX−Variante).

energy spezifiziert, wieviel Energie für diese Operation auf dem Marktplatz geboten werden soll. Ein Wert von −1 bedeutet „soviel, wie verfügbar“. Dadurch wird entweder der Wert des Prozesskontos oder − falls vorhanden − der Wert des Dateikontos (siehe open2) verwendet. Ein Angabe für *energy* größer oder gleich Null beschränkt das Gebot für diesen Aufruf zusätzlich. *energy* muss kleiner oder gleich der maximal verfügbaren Energie (Prozess− oder Dateikonto, siehe oben) sein.

timeout spezifiziert den Timeout in Sekunden (wie open2).

Die POSIX−Variante entspricht den Parametern *energy = −1* und *timeout = −1*.

Syntax:

```
ssize_t write2(   int d, void *buf, size_t nbytes,
                  int energy, int timeout);
```

Versucht Daten auf eine Datei zu schreiben (Parameter *d*, *buf*, *nbytes* wie POSIX−Variante). Es gelten die Erläuterungen zu read2.

Weitere Aufrufe:

Syntax:

```
int sync2(int energy, int timeout);
```

Leert die Puffer des Filesystems und und schreibt geänderte Informationen auf die Platte (wie POSIX−Variante).

energy spezifiziert, wieviel Energie für diese Operation auf dem Marktplatz geboten werden soll. Ein Wert von −1 bedeutet „soviel, wie verfügbar“. Dadurch wird wieder der Wert des Pro−zesskontos verwendet. Ein Wert größer oder gleich Null beschränkt das Gebot für diesen Aufruf. *energy* muss kleiner oder gleich der maximal verfügbaren Energie auf dem Prozesskonto sein.

timeout spezifiziert den Timeout in Sekunden (wie open2).

Die POSIX−Variante entspricht den Parametern energy = −1 und timeout = −1.

Über die hier vorgestellten erweiterten Aufrufe hinaus wurden alle auf Systemaufrufe, die Zu−griffe auf die Festplatte auslösen können, so geändert, dass vorher Gebote in Höhe des Prozesskontos auf dem Marktplatz abgegeben werden. Auf Dateideskriptoren arbeitenden Sys−temaufrufe wurden so modifiziert, dass sie ein vorhandenes Dateikonto respektieren und dessen Wert als Gebot abgeben. Im geänderten MINIX−System ist somit kein Zugriff auf die Festplatte ohne Gebot und Bezahlung am Marktplatz möglich.

4.2.3. Neu geschaffene Systemaufrufe

Zusätzlich zu den eben vorgestellten, existiert eine Reihe von neuen Aufrufen. Diese dienen der Verwaltung der Energiekonten oder der Steuerung der beteiligten Energiesparkomponenten im Betriebssystem. Um die Änderungen in Grenzen zu halten und die Übersichtlichkeit zu fördern, sind alle neu geschaffenen Aufrufe nur verschiedene Parametrierungen eines einzigen echten Systemaufrufs, des POWERCTL−Aufrufs.

Anmerkung: Um die folgenden Systemaufrufe nutzen zu können, muss die Headerdatei <minix/powerctl.h> ein−gezogen werden. Die Implementierung ist in die Standard−C−Bibliothek eingebunden.

Verwaltung der Energiekonten:

Syntax:

```
int getenergy();
```

Liefert den aktuellen Wert des Prozesskontos zurück.

Syntax:

```
int getfenergy(int d);
```

Liefert den aktuellen Wert des Dateikontos für Dateideskriptor *d* zurück. Existiert kein Datei−konto liefert der Aufruf −1.

Syntax:

```
int setfenergy(int d, int energy);
```

Setzt den Wert des Dateikontos für Dateideskriptor *d* auf *energy*. Existiert kein Dateikonto, so wird es angelegt. Dieser Aufruf dient dazu, Energie zwischen Prozesskonto und Dateikonto zu transferieren. Eine Wert für *energy* von −1, löst das Dateikonto auf. *energy* kann nie größer sein als die Summe von Prozess− und Dateikonto. Nach erfolgreicher Ausführung wird Null zurückgegeben.

Syntax:

```
int addfenergy(int d, int energy);
```

Erhöht den Wert des Dateikontos für Dateideskriptor *d* um *energy*. *energy* kann auch nega−tiv sein. Existiert kein Dateikonto, so wird −1 zurückgegeben. Dieser Aufruf dient wie setfenergy dazu, Energie zwischen Prozesskonto und Dateikonto zu transferieren. Er ist aus Komfort− und Performance−Gründen aufgenommenen worden. Nach erfolgreicher Ausführung wird Null zurückgegeben.

Soweit die Aufstellung der Systemaufrufe, die Benutzerprozessen die Möglichkeit geben, ihren Energieverbrauch für Festplattenzugriffe zu beschränken. Wie effizient und intelligent diese Aufrufe eingesetzt werden, bleibt dem Entwickler des Anwendungprogrammes überlassen. In Kapitel 4.4 werden Beispiele vorgestellt, wie durch spezielle Algorithmen beim Lesen oder Schreiben von Dateien Energie eingespart werden kann.

Reservierte Systemaufrufe:

Die folgenden Aufrufe sollten sinnvollerweise Programmen, die dem Superuser gehören, vor−behalten sein, da sie teilweise das Energiesparkonzept umgehen und aushebeln. Sie bieten die Möglichkeit, direkt mit Komponenten in den unteren Schichten zu kommunizieren und sind

daher eher für die Fehleranalyse interessant. Um in der Testphase einen unkomplizierten Zugriff und schnelle Diagnose zu ermöglichen, sind sie im vorliegenden System jedoch allgemein zugänglich.

Syntax:

```
int powerctl(int mode);
int powerctl2(int mode, int value);
```

Diese beiden Interfaces bieten Zugriff auf eine Reihe von speziellen Möglichkeiten. Der Parameter mode hat folgende Bedeutung (P_ steht jeweils für powerctl):

mode	***Funktion***
P_INQUIRE	Liefert zurück, ob der Spindel−Motor der Festplatte läuft (1) oder nicht (0).
P_IDLE	Versetzt die Platte sofort in den Idle−Mode.
P_STANDBY	Versetzt die Platte sofort in den Standby−Mode.
P_SLEEP	Versetzt die Platte sofort in den Sleep−Mode.
P_GETCOSTS	Befragt den Modusverwalter nach dem aktuellen Preis.
(P_PUTENERGY)	(Für Debugging: Erzeugt eine Nachricht, mit der das Filesystem Energie an den Modusverwalter transferiert.)
P_GETPDTIME	Liefert den Standby−Timeout.
P_SETPDTIME	Setzt den Standby−Timeout (powerctl2−Interface verwenden). Ein Wert von −1 deaktiviert den Standby−Timeout.
P_CLOCKTICK	Sendet ein Zeitsignal an Filesystem und Modusverwalter.

Im laufenden Betrieb werden unter normalen Umständen nur die letzten drei Modi verwendet. Die Variation des Standby−Timeouts kann dazu verwendet werden, einen optimalen Wert für die vorliegende Konfiguration zu finden. *P_CLOCKTICK* ist der einzige Modus, der für den Betrieb des Energiesparkonzepts essentiell ist. Der Clocktick−Prozess benutzt ihn, um das Zeitsignal zu versenden. Dieser Aufruf wird im Sekundentakt getätigt und dient als fortlaufende Uhr. Auf die Implementierung wird in den Kapiteln über die entsprechenden Komponenten eingegangen.

Nachdem die Sicht der Benutzerprozesse auf das Energiesparkonzept dargestellt ist, sollen die darunter liegenden Komponenten im Betriebssystem näher betrachtet werden.

4.2.4. Filesystem und Marktplatz

Überblick

Das Filesystem ist die umfangreichste der beteiligten Komponenten. Es handelt sich dabei um eine Art Fileserver. Die obere Schnittstelle bilden die auf Dateien arbeitenden Systemaufrufe, darunter auch die eben vorgestellten. Sie werden von den Benutzerprozessen durch das Versenden von Nachrichten an das Filesystem getätigt. Entsprechende Stubs in der C−Bibliothek übernehmen dabei, wie dargelegt, das Marshalling. Die untere Schnittstelle bilden eine Reihe

von Kommandos, die der Modusverwalter akzeptiert, hauptsächlich sind dies Festplattenkommandos.

Aufgabe des Filesystems ist es, alle Systemaufrufe, die den Umgang mit Dateien betreffen, auszuführen. Dies umfasst die Behandlung der gängigen Systemaufrufe (wie read, open, ...). Dafür existiert pro Systemaufruf eine spezielle Behandlungsroutine. Ebenso müssen aber grundlegende Aufgaben, wie die Verwaltung des Dateisystems (MINIX−FS) und Caching−Mechanismen behandelt werden. Das Filesystem umfasst daher eine Ansammlung von Funktionen, die diese Aufgaben erledigen und als Bibliothek für die spezifischen Behandlungsroutinen dienen. Abbildung 16 gibt einen Überblick über die Bearbeitung eines Systemaufrufs durch das Filesystems.

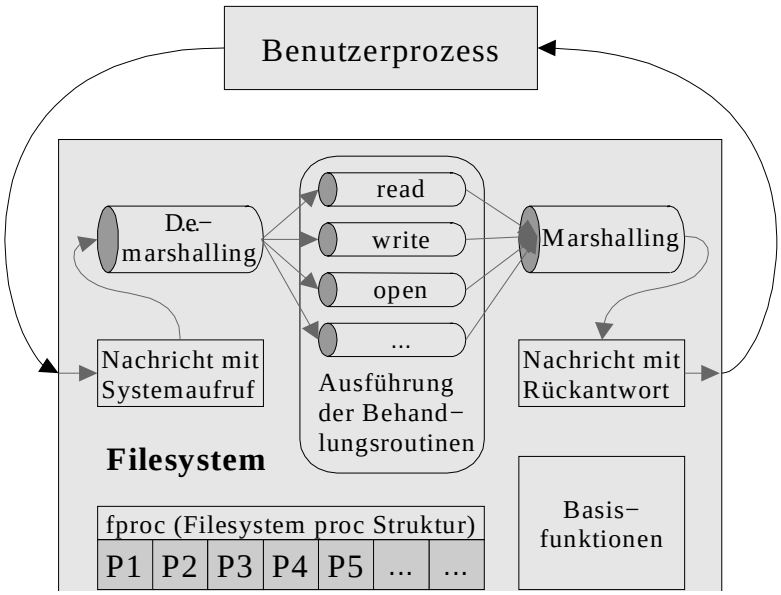

Abbildung 16: Bearbeitung eines Systemaufrufs im Filesystem

Problematik des fehlenden Transaktionskonzepts

Beim Versuch, das Konzept der kooperativen Nutzung aus Kapitel 3 umzusetzen, tritt ein grundlegendes Problem auf. Das Grundkonzept, Preise für Festplattenoperationen auf dem Marktplatz auszuhandeln und Energie entsprechend abzurechnen, geht davon aus, dass entschieden werden kann, ob ein Systemaufruf einen Zugriff auf die Festplatte nötig macht. Davon hängt der Preis eines Systemaufrufs ab. Kann der Aufruf ohne Festplattenzugriff bearbeitet werden, ist er kostenlos, und es muss kein Gebot am Marktplatz abgegeben werden.

Dies festzustellen, ist jedoch aus mehreren Gründen problematisch. Auf Dateien arbeitende Aufrufe müssen nicht zwingend auf einer Festplattendatei arbeiten. Es kommen beispielsweise auch Terminals, Netzwerk−Verbindungen oder Gerätedateien in Frage. Auch wenn geklärt werden kann, dass es sich tatsächlich um eine Festplattendatei handelt, bleibt offen, ob ein Zugriff auf den Datenträger durch das Caching verzögert wird oder überhaupt nicht erfolgt.

Diese Problematik ließe sich zunächst dadurch umgehen, dass ein Gebot auf dem Marktplatz erst dann abgegeben wird, wenn feststeht, dass ein Festplattenkommando an den Modusverwalter

verschickt wird, also direkt vor dem Versenden der Nachricht. Doch auch diese Lösung ist nicht praktikabel. Der Hauptgrund dafür liegt im programmiertechnischen Aufbau des Filesystems und dem Fehlen eines Transaktionskonzepts. An dieser Stelle wurden ursprünglich nur Festplatten−kommandos an die Disk Task verschickt. Es existiert jedoch keine sinnvolle Fehlerbehandlung für die Antwortnachrichten. Offensichtlich wurden Lese−/Schreibfehler auf dem Medium für zu selten erachtet. Somit kann ein Aufruf, der durch ein zu geringes Energiegebot nicht ausgeführt werden soll (das Gebot auf dem Marktplatz ist nach Ablauf des Timeout verfallen), an dieser Stelle nur mit großem Aufwand abgebrochen werden, da ein Abbruch nicht vorgesehen ist. Es müssten mit vielen Fallunterscheidungen alle bisher getätigten Änderungen wieder zurückge−nommen werden, um die interen Verwaltungsstrukturen in einem konsistenten Zustand zu hinterlassen. Dieses nachträgliche Einbringen eines Transaktionskonzepts würde in Teilen einer Neuimplementierung des Filesystems gleichgekommen und hätte den Rahmen dieser Arbeit ge−sprengt.

Die Lösung aus diesem Dilemma kann nur ein Kompromiss sein. In Fällen, in denen eindeutig entschieden werden kann, dass kein Festplattenzugriff notwendig ist, darf der Marktplatz um−gangen werden und der Aufruf ohne weitere Beachtung des Energiesparkonzepts ausgeführt werden. Ansonsten muss generell ein Gebot auf dem Marktplatz abgegeben werden. Denn es könnte eine kostenpflichtige Festplattenoperation getätigt werden (worst−case−Annahme).

Auf diese Weise wird zwar ein Teil der Aufrufe zu Unrecht mit Kosten belegt, dies kann aber in Kauf genommen werden. In den späteren Testapplikationen kommt nur eine kleine Auswahl der Systemaufrufe zum Einsatz (read, write, open, close, sync). Bei den Behandlungsroutinen dieser Aufrufe wurde darauf geachtet, dass der überwiegende Teil korrekt mit Kosten belegt wird. Ohnehin geht dadurch keine Energie global verloren, der betroffene Prozess steuert nur zu den Kosten einer Aktion, die andere Prozesse ausführen, einen Teil bei. In einem Konzept, das sich „kooperativ“ nennt, ist dies nicht unbedingt falsch, zumal der Fehler keinen Prozess einseitig benachteiligt. Für eine zukünftige, fehlerfreie Umsetzung sind die oben erwähnten umfangrei−chen Änderungen am Filesystem unabdingbar.

Folgende Modifizierungen wurden am Filesystem durchgeführt:

1. Der Marktplatzes wurde eingebettet (vergleiche Abbildung 14). Dies ist notwendig, da der Marktplatz seine Aufgaben ohne einen direkten Zugriff auf Datenstrukturen des Filesystems, inbesondere des Prozesskontextes (fproc) nur sehr ineffizient erfüllen könnte.
2. Die Behandlungsroutinen der Systemaufrufe wurden dahingehend geändert, dass vor einem möglichen Festplattenzugriff zuerst ein Gebot auf dem Marktplatz abgegeben wird. Nur wenn erfolgreich ein Preis ausgehandelt ist, wird mit der Ausführung fortgefahren.
3. Die Behandlungsroutinen wurden ergänzt, damit die neu hinzugekommenen Systemaufrufe (siehe Kapitel 4.2.3) ausgeführt werden können.

Ablauf bei der Bearbeitung eines Systemaufrufs

Abbildung 17 gibt eine Übersicht über den grundsätzlichen Ablauf eines Aufrufs an das File−system. Dabei liegt der Fokus auf der oberen Schnittstelle, also auf der Kommunikation mit dem Benutzerprozess.

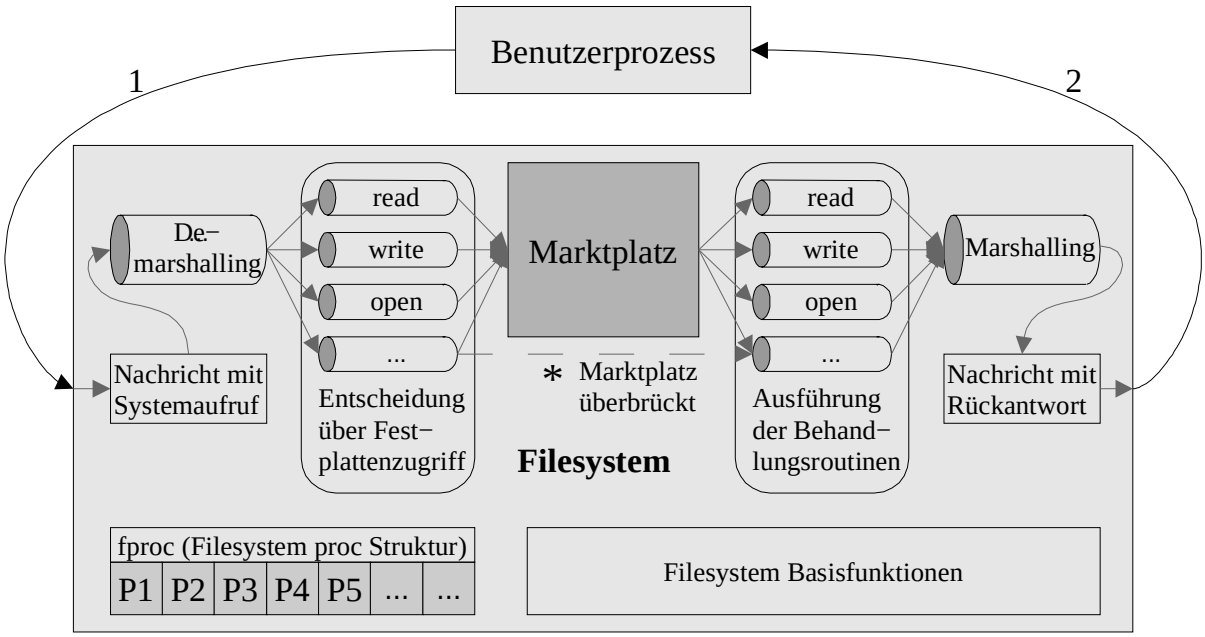

Abbildung 17: Bearbeitung eines Systemaufrufs im erweiterten Filesystems

Der Benutzerprozess tätigt einen Systemaufruf, indem er eine Nachricht mit der Aufrufkennzahl und den Parametern an das Filesystem verschickt. Nach dem Demarshalling wird die passende Behandlungsroutine für diesen Aufruf angesprungen. Diese implementiert die Funktionalität des Systemaufrufs. Dabei kann auf eine breite Palette von Basisfunktionen zurückgegriffen werden wie zur Deskriptorverwaltung und dem Caching. Die Basisfunktionen sind teilweise hierarchisch gegliedert. Ein Zugriff auf die Festplatte erfolgt nur an einer zentralen Stelle durch Versenden einer Nachricht an den Modusverwalter.

In den Behandlungsroutinen wird jetzt entschieden, ob dieser Aufruf Festplattenzugriffe benötigt oder nicht. Diese Entscheidung ist nicht unproblematisch, es reicht jedoch, hier die wichtigsten Fälle auszusondern, bei denen *kein* Festplattenzugriff notwendig ist. In diesem Fall kann der Marktplatz überbrückt (*) und direkt mit der Ausführung begonnen werden. Als Beispiel sei hier der write−Aufruf auf ein Terminal zu nennen. Da unter MINIX kein Debugger zur Verfügung steht, wird dieser Aufruf oft zur Ausgabe von Debugging−Informationen gebraucht, und soll selbstverständlich weder aus „Energiemangel“ abgebrochen oder verzögert werden, noch in die Gebote auf dem Marktplatz eingreifen.

Ist ein Festplattenzugriff nicht auszuschließen, so wird der Marktplatz angesprungen und ein entsprechendes Gebot platziert. Auf die genauen Abläufe im Marktplatz wird gleich eingegan−gen. Kehrt der Aufruf zurück, so signalisiert der Rückgabewert, welcher der folgenden beiden Fälle aufgetreten ist:

1. Der Aufruf wurde blockiert, weil nicht genug Energie geboten worden ist.
2. Es steht genug Energie zur Verfügung.

Im ersten Fall wird die Ausführung abgebrochen und auf einen neuen Aufruf gewartet. Ansons−ten wird der Systemaufruf endgültig bearbeitet und das Ergebnis an den Benutzerprozess zurückgeschickt. Wie in Abbildung 15 angedeutet, muss dazu unter Umständen die System Task beauftragt werden, da nur sie Daten in den Benutzeradressraum kopieren kann. Die Kommuni−kation mit der System Task wird im Folgenden vernachlässigt, um den Blick auf das Wesentliche zu konzentrieren.

Platzieren eines Gebots auf dem Marktplatz

Der Marktplatz ist keine eigenständige Komponente, sondern in das Filesystem eingebettet. Seine Aufgabe ist es, Gebote der Benutzerprozesse zu sammeln, bis genug Energie zusammengekommen ist, um die gewünschte Operation auszuführen. Seine Funktion gleicht damit einem Vermittler zwischen den energetischen Preisvorstellungen der Benutzerprozesse und denen des Betriebssystems.

Bedingt durch die Einbettung ist echter Nachrichtenverkehr zwischen Marktplatz und Filesystem überflüssig. Der Marktplatz besteht aus einer Reihe von Funktionen, von denen zwei als Einsprungpunkte aus den Behandlungsroutinen des Filesystems heraus dienen. Dabei wird die ursprünglich vom Benutzerprozess an das Filesystem gesendete Nachricht mit an den Marktplatz übergeben. Man kann somit dennoch von einer Art „Nachrichtenverkehr“ reden. Sind die bisherigen Gebote in ihrer Summe zu wenig, um ausgeführt werden zu können, so müssen diese verzögert werden. Dazu ist ein Puffer notwendig, in dem die blockierten Nachrichten gehalten werden.

Der Marktplatz besitzt aus Sicht des Filesystems zwei Funktionen:

1. Eine Funktion zum Platzieren eines Gebots (`check_energy`)
2. Eine Funktion zum Reaktivieren blockierter Aufrufe (`blocked_message`)

Die Funktion `check_energy` dient zum Platzieren eines Gebots. Dabei müssen einige Bedingungen überprüft werden. Abbildung 18 zeigt die notwendigen Abläufe.

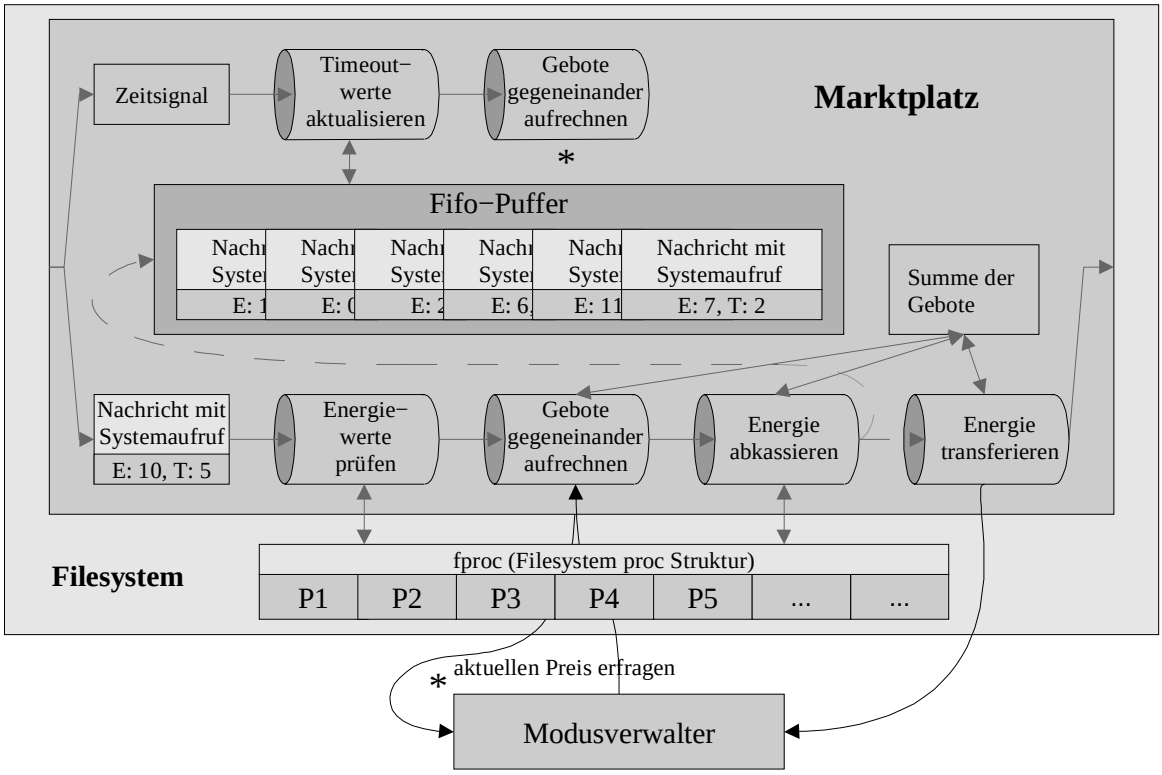

Abbildung 18: Das Platzieren eines Gebots auf dem Marktplatz

Soll ein Gebot für einen Systemaufruf mittels *check_energy* platziert werden, so muss in einem ersten Schritt überprüft werden, ob dies zulässig ist (unterer Ast in Abbildung 18). Ein Gebot kann nie höher sein, als die zur Verfügung stehende Energie. Anderenfalls wird es nach unten korrigiert. Welches Konto für diesen Systemaufruf zu verwenden ist, wird bereits im Filesystem entschieden. Anhand des Aufrufers (Absenders der Nachricht) wird der entsprechende Eintrag im Prozesskontext des Filesystems selektiert. Die Behandlungsroutine entscheidet darüber hinaus, ob statt dem Prozesskonto ein Dateikonto zu belasten ist.

Nachdem die Zulässigkeit des Gebots verifiziert ist, müssen die Gebote der Benutzerprozesse gegen das Gebot des Modusverwalters aufgerechnet werden. Alle bisher eingegangenen Systemaufrufe werden in einem Fifo−Puffer gehalten. Die Summe der Gebote ist bereits berechnet worden. Da sich das Gebot des Modusverwalters zeitlich ändert, muss ein aktuelles Gebot eingeholt werden. Dazu sendet der Marktplatz eine Preisanfrage an den Modusverwalter. Liegt jetzt der aktuelle Preis vor, so gibt es zwei Möglichkeiten:

1. Die Gesamt−Energie der Gebote übersteigt den vom Modusverwalter geforderten Preis.
2. Die Gesamt−Energie der Gebote ist noch zu gering.

Im ersten Fall wird der Wert des Gebots vom Konto des Aufrufers abgebucht und die Summe der gebotenen Energie dem Modusverwalter überschrieben. Dazu wird eine spezielle Nachricht, die Transferbuchung verwendet. Damit gilt der geforderte Preis als bezahlt und der Systemaufruf kann unmittelbar ausgeführt werden. Anschließend werden alle blockierten Aufrufe im Puffer deblockiert und über die *blocked_message* Funktion, nach dem Fifo−Prinzip reaktiviert. Eine Beschreibung dieses Mechanismus folgt noch.

Im zweiten Fall steht nach wie vor nicht genügend Energie zur Verfügung. Daher muss der Aufruf blockiert werden. Dazu wird ebenfalls die gebotene Energie bereits abgebucht, damit sie nicht zu einem weiteren Gebot zur Verfügung steht, während dieser Aufruf blockiert ist. Dies wäre möglich, wenn ein Vaterprozess und sein Kindprozess auf einem gemeinsamen Dateideskriptor arbeiten, der ein Dateikonto beinhaltet. Anschließend wird die zugehörige Nachricht in den Fifo−Puffer eingereiht und die Summe der Gebote aktualisiert. Der Behandlungsroutine im Filesystem wird per Rückgabewert signalisiert, dass der Aufruf blockiert wurde.

Realisierung des Timeouts

Zuletzt stellt sich die Frage, wie der Timeout realisiert wird. Dazu wird auf das vom extra eingeführten Clocktick−Prozess versendete Zeitsignal zurückgegriffen. Dieses Zeitsignal kann mit dem *powerctl* Aufruf erzeugt werden. Es gleicht einem gewöhnlichen Systemaufruf, Aufrufnummer und Parameter lassen darauf schließen, dass es sich um die *P_CLOCKTICK* Variante des *powerctl* Aufrufs handelt. Beim Eintreffen einer Nachricht dieses Typs wird in den oberen Ast von Abbildung 18 verzweigt. Als Reaktion wird der Timeout aller Nachrichten im Puffer um eine Sekunde verringert. Fällt ein Timeout auf Null, so wird der betreffende Systemaufruf beim nächsten Aufruf von *blocked_message* erkannt und eine Nachricht mit *ENOENERGY* an den Benutzerprozess zurückgesendet.

Darüber hinaus wird bei dieser Gelegenheit ein neues Gebot vom Modusverwalter eingeholt. Es könnte sein, dass der Preis soweit gesunken ist, dass die bisher gebotene Energie bereits ausreicht, um ihn zu bezahlen. In diesem Fall wird vermerkt, dass der Puffer gelehrt werden kann.

Der nächste *blocked_message* Aufruf wird dann die blockierten Nachrichten nach dem Fifo−Prinzip reaktivieren.

Reaktivierung blockierter Aufrufe

Die Funktion *blocked_message* reaktiviert blockierte Aufrufe. Sie testet also, ob ein blockierter Aufruf existiert, der entweder jetzt bearbeitet werden kann, oder ob dessen Geltungsdauer (Timeout) abgelaufen ist. In beiden Fällen wird der Aufruf reaktiviert und kann sofort bearbeitet oder mit der Fehlermeldung *ENOENERGY* abgewiesen werden. *blocked_message* wird deshalb im Filesystem grundsätzlich vor dem (blockierenden) Empfangen neuer Nachrichten von Benutzerprozessen ausgeführt. Nur wenn kein (alter) Systemaufruf reaktiviert werden kann, wird auf das Eintreffen neuer Aufrufe gewartet. Auf diese Weise werden deblockierte Systemaufrufe sofort bearbeitet, bevor weitere Aufrufe entgegengenommen werden.

Kommunikation mit dem Modusverwalter

Nach dieser Betrachtung der grundsätzlichen Abläufe im Filesystems, soll jetzt das Augenmerk auf die Schnittstelle zum Modusverwalter gelegt werden. Wie bereits erläutert, verwendet der Marktplatz zwei Typen von Nachrichten für die Kommunikation mit dem Modusverwalter:

1. Preisanfrage
2. Transferbuchung

Neben dem Marktplatz kommunizieren auch die Basisfunktionen des Filesystems mit dem Modusverwalter. Es werden vier Nachrichtentypen verwendet:

1. Disk−Kommando
2. Preisanfrage (als Reaktion auf die *P_GETCOSTS*−Variante des powerctl−Systemaufrufs, siehe letztes Kapitel)
3. Zeitsignal (als Reaktion auf die *P_CLOCKTICK*−Variante des powerctl−Systemaufrufs)
4. Steuernachricht (als Reaktion auf die anderen Varianten des powerctl−Systemaufrufs)

Disk−Kommandos sind dabei der mit Abstand häufigste Nachrichtentyp. Es handelt sich hauptsächlich um Schreib−/Leseanforderungen von Blöcken, die der Modusverwalter an die Disk Task weiterleitet. Die drei verbleibenden Typen (Preisanfrage, Zeitsignal und Steuernachricht) sind seltener. Vom Filesystem erzeugte Preisanfragen und Steuernachrichten sind hauptsächlich für Debugging−Zwecke interessant, Zeitsignale treffen, wie bekannt, einmal pro Sekunde ein, um Zeitmessungen zu ermöglichen.

Während Preisanfragen, Zeitsignale und Steuernachrichten direkte Konsequenzen eines Systemaufrufs sind (nämlich des *powerctl*−Aufrufs in einer seiner Varianten), werden Disk−Kommandos nur dann abgesetzt, wenn ein Systemaufruf nicht anderweitig, beispielsweise aus dem Cache, bearbeitet werden kann.

4.2.5. Modusverwalter

Der Modusverwalter ist ebenfalls eine der zentralen Komponenten des Energiesparkonzepts. Seine Aufgabenspektrum umfasst:

1. Treffen von Entscheidungen über einen Powerdown oder Spinup
2. Festlegen der Preise für Aktionen
3. Abrechnen des Basisenergieverbrauchs
4. Abrechnen des Energieverbrauchs bei Aktionen
5. Weiterleiten des Nachrichtenverkehrs zwischen Filesystem und Disk Task

Um den Energieverbrauch korrekt abzurechnen, werden zwei Energiekonten benötigt: das Sys−temkonto und das Modusverwalterkonto. Das Systemkonto wurde bereits in Kapitel 3 erwähnt und dient zur Abrechnung des Basisenergieverbrauchs. Das Modusverwalterkonto dient als Übertragskonto. Ist auf dem Marktplatz eine Einigung erzielt worden, „kassiert“ der Marktplatz die gebotene Energie, bucht sie von den Prozess− oder Dateikonten ab und transferiert sie zum Modusverwalter.

Wie aus Abbildung 15 zu sehen ist, kommuniziert der Modusverwalter nur mit dem Filesystem und der Disk Task. Seine Funktionsweise kann erklärt werden, indem die Reaktionen auf ein−treffende Nachrichten dargestellt werden. Es sind zwei Fälle zu unterscheiden:

1. Nachrichten vom Filesystem und
2. Nachrichten von der Disk Task.

Der zweite Fall ist einfach. Es handelt sich stets um Bestätigungen für angeforderte Festplat−tenoperationen. Sie werden an das Filesystem weitergeleitet.

Abbildung 19 gibt eine Überblick über die Reaktionen auf Nachrichten, die vom Filesystem eintreffen (1. Fall).

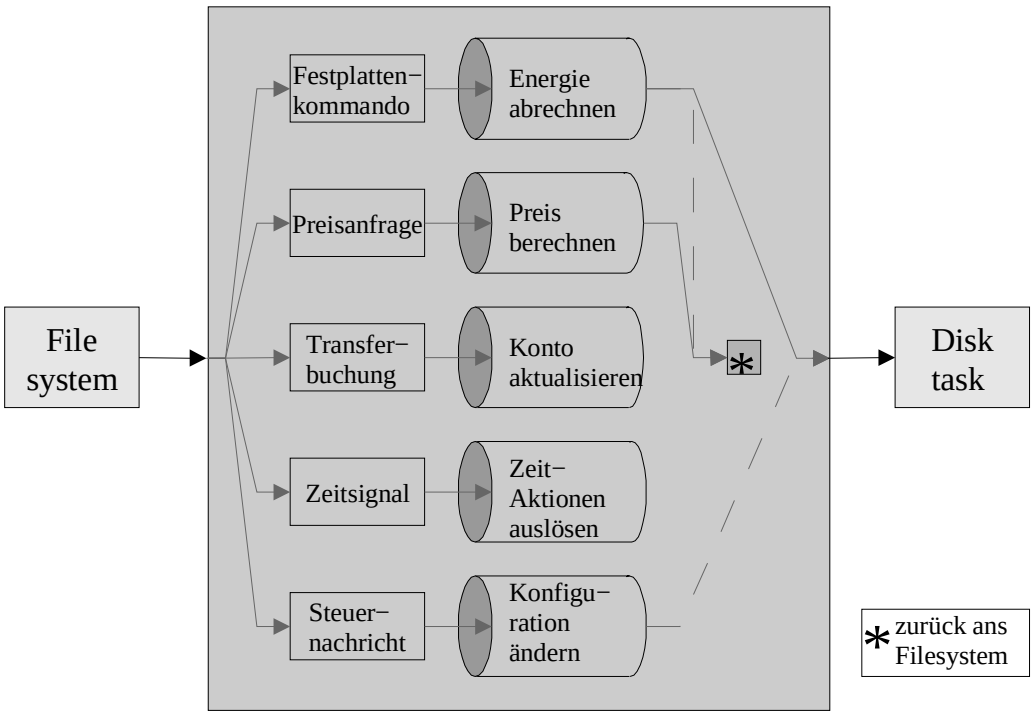

Abbildung 19: Reaktionen auf Nachrichten vom Filesystem

Kursivschrift soll im Folgenden die Zuordnung zwischen den Nachrichtentypen und den Aufgaben des Modus−verwalters erleichtern.

Beim ersten Nachrichtentyp handelt es sich um *Festplattenkommandos*, die für die Disk Task bestimmt sind. Diese werden *an die Disk Task weitergeleitet*. Der Grund, warum diese Nach−richten überhaupt zum Modusverwalter geschickt werden, liegt darin, dass Nachrichten innerhalb des Filesystems uniform behandelt werden. Eine Unterscheidung der Nachrichtentypen erfolgt im Filesystem nicht. Auch ist damit der Modusverwalter die einzige Komponente, die Kom−mandos an die Disk Task verschickt. Es ist somit leichter für den Modusverwalter auf dem aktuellen Stand zu bleiben, was den Energiemodus der Festplatte betrifft, da keine Kommandos von anderen Komponenten unbemerkt einen Spinup auslösen können. Anfragen an die Disk Task bezüglich des Modus der Festplatte können damit auf ein Minimum reduziert werden.

Gleichzeitig muss die *Abrechnung des Energieverbrauchs für die Aktionen* erfolgen. Passiert ein Festplattenkommando den Modusverwalter, wird vom Modusverwalterkonto der Preis für diese Aktion abgezogen und ein Flag gesetzt. Treffen weitere Kommandos ein, sind sie kostenlos, so−lange das Flag gesetzt ist. Beim Eintreffen eines Zeitsignal wird das Flag wieder zurückgesetzt. Auf diese Weise wird dafür Sorge getragen, dass für alle Systemaufrufe, die an der Einigung auf dem Marktplatz beteiligt waren, nur einmal die Kosten abgerechnet werden. Da das Filesystem neue Nachrichten erst annimmt, nachdem alle deblockierten Aufrufe abgearbeitet sind, kann ein Zeitsignal diese Aufrufe nicht „überholen“.

Preisanfragen werden hauptsächlich vom Marktplatz innerhalb des Filesystems verschickt, um das Gebot des Modusverwalters einzuholen. Der Modusverwalter gibt also nicht selbst Gebote ab, sondern wird vom Marktplatz regelmäßig nach dem aktuellen Preis gefragt. Vom eigentli−chen Preis wird stets der Stand des Modusverwalterkontos abgezogen, der Preis sinkt aber selbstverständlich nicht unter Null. Somit beträgt nach einer Transferbuchung der Preis solange Null, bis der durch die Transferbuchung geleistete „Vorschuss“ aufgebraucht ist. Erst dann sind die Aktionen wieder kostenpflichtig. Die *Berechnung des Preises* wurde theoretisch bereits in Kapitel 3.4.2 erläutert. Die konkrete Implementierung der dort genannte Formel kann nicht mehr allgemein gehalten werden, sondern muss auf den realen Energieverbrauch abgestimmt sein (Problematik der Rundungsfehler). Daher wird sie im Zuge der Parametrierung des Systems in Kapitel 4.3 vorgestellt.

Transferbuchungen sind ebenfalls Nachrichten des Marktplatzes. Hierdurch wird Energie vom Marktplatz auf das Modusverwalterkonto gebucht.

Das *Zeitsignal*, das jeweils im Abstand von einer Sekunde eintrifft, wird benötigt für die zeit−abhängigen Preisberechnungen im Standby Mode. Die Implementierung des Standby Timeouts ist ebenfalls an dieses Signal geknüpft, wie auch die Abrechnung des Basisenergieverbrauchs. Es wird auf dem Marktplatz bereits für Timeoutmessungen benötigt und an den Modusverwalter weitergegeben. Der Standby Timeout ist sehr einfach zu implementieren. Der Standby−Zähler wird durch jedes eingetroffene Zeitsignal erhöht. Ein Festplattenkommando setzt ihn dagegen auf Null. Steigt der Wert über den Standby Timeout, so erfolgt der *Powerdown*. Die *Abrechnung des Basisenergieverbrauchs* erfolgt dadurch, dass beim Eintreffen eines Zeitsignals der dem ak−tuellen Modus entsprechende Verbrauch vom Betriebssystemkonto abgezogen wird.

Schließlich können noch *Steuernachrichten* eintreffen. In diese Kategorie fallen Nachrichten zum Lesen und Setzen des Standby−Timeouts, sowie zum expliziten Wechseln in andere Ener−giemodi. Diese Funktionalität wurde vor allem für die in Kapitel 2.2 angestellten Messungen benötigt.

4.2.6. Disk Task

Die Disk Task fungiert in MINIX als Festplattentreiber. Bei genauerer Betrachtung fällt auf, dass es sich hierbei um mehrere Tasks handelt, die nicht nur Festplatten verwalten. Sie arbeiten teilweise auf gemeinsamen Funktionen, sind jedoch sonst unabhängig. Auf diese Art werden die verschiedenen Typen von Controllern (IBM−AT, Adaptec SCSI, Floppy, Mitsumi CD−Rom, ...) von unterschiedlichen Tasks verwaltet, aber über ein gemeinsames Interface angesprochen. Für die in dem später vorgestellten Testsystem eingesetzte AT−kompatible Festplatte ist die soge−nannte Winchester−Task zuständig.

Alle Tasks führen eine gemeinsame Hauptschleife aus, in der eine eingegangene Nachricht ent−schlüsselt und die entsprechende Funktion aufgerufen wird. Das gemeinsame Interface stellt unter anderem Funktionen wie *dr_open*, *dr_prepare*, *dr_geometry*, *dr_schedule* oder *dr_finish* zur Verfügung. Damit kann ein an den Controller angeschlossenes Gerät ini−tialisiert und zum Datenaustausch vorbereitet werden. Es können Informationen über das Gerät abgefragt sowie Lese− oder Schreiboperationen angestoßen werden. Eine Kontrolle der Ener−giemodi ist mit diesen Funktionen allerdings nicht möglich.

Deshalb wird das Interface um eine Funktion erweitert: *dr_powerctl*. Die Funktion besitzt einen numerischen Parameter und kann sowohl zum Wechseln als auch, soweit möglich, zur Abfrage des momentanen Energiemodus verwendet werden. Da jede Task das gesamte Interface implementieren muss, ist eine leere Funktion mit diesem Namen notwendig. Bis auf die Win−chester−Task verwenden alle Tasks diese Dummy−Funktion.

In der Winchester−Task kann diese Funktion sinnvoll implementiert werden. Abbildung 20 zeigt die gültigen Parameter und Rückgabewerte.

```
int dr_powerctl(int mode);

mode:

P_INQUIRE  Energiemodus abfragen
P_IDLE     in Idle Mode wechseln
P_STANDBY  in Standby Mode wechseln
P_SLEEP    in Sleep Mode wechseln

Rückgabewert:

P_INQUIRE  0 (Standby, Sleep) 1 sonst
sonst      OK oder Fehlerwert
```

Abbildung 20: Syntax des Powerctl−Aufrufs des Festplattentreibers

Die Winchester−Task implementiert diese Funktionalität, indem sie die zugehörigen ATAPI−Kommandos in die Register des IBM−AT Controllers schreibt. Zur Abfrage des Energiemodus ist anzumerken, dass es nicht möglich ist, festzustellen, in welchem Modus sich die Platte be−findet. Es existiert jedoch ein ATAPI−Kommando, mit dem überprüft werden kann, ob der Spindel−Motor läuft. Daher kann nur die Entscheidung (Active, Idle) oder (Standby, Sleep) ge−troffen werden. Darüber hinaus ist wissenswert, dass der ATAPI−Standard nicht zwischen Performance Idle Mode und Low Performance Idle Mode unterscheidet. Messungen haben aber

ergeben, dass die verwendete Platte von IBM (DTNA−22160) bei einem Kommando zum Wechsel in den Idle Mode in den Performance Idle Mode wechselt.

Der Wechsel in den Standby Mode muss nicht durch ein explizites Kommando erfolgen, sondern kann auch nach einem Timeout von der Platte selbst eingeleitet werden. Da eine platteneigene Entscheidung aber unerwünscht ist und sich der Timeout nicht vollständig deaktivieren lässt, setzt die Winchester Task beim Initialisieren des Controllers den Standby Timeout auf den höchstmöglichen Wert von 109 Minuten. Dies stellt sicher, dass vorher der MINIX−eigene Standby−Timeout greift, da dieser bei sinnvoller Konfiguration wesentlich kürzer ist. In diesem Fall wird die Platte dann durch ein explizites Wechselkommando in den Standby Mode gebracht.

4.2.7. Zusammenspiel der Komponenten

Das Zusammenspiel der oben erklärten Komponenten kann anhand eines read−Systemaufrufs erläutert werden. Dazu werden die wichtigen Abläufe schrittweise erklärt. Die genannten Energiewerte sind vorerst als hypothetisch anzusehen. Die Parametrierung des Systems wird erst in Kapitel 4.3 erläutert.

Ausgangsszenario

Um möglichst alle Facetten des Konzeptes zeigen zu können, soll der read−Aufruf auf eine Festplattendatei erfolgen, die vorher mit

```
fd = open2("test.txt“, O_RDONLY, 150, −1);
```

erfolgreich geöffnet worden ist. Dies bedeutet, dass 15 Joule für das Anlegen der Datei zur Verfügung standen bei deaktiviertem Timeout, die Restenergie wurde auf dem Dateikonto deponiert. Mit

```
e = getfenergy(fd);
```

wurde ermittelt, dass nach dem Anlegen noch 9,3 Joule übrig waren.

Der jetzt folgende Aufruf soll sein:

```
n = read2(fd, &test_data, sizeof(test_data), 20, 3);
```

Für den read−Aufruf sollen also maximal 2 Joule verwendet werden, der Timeout beträgt 3 Sekunden.

Ablauf des Aufrufs

Da eine vollständige Vorstellung aller möglichen Konstellationen nicht möglich ist, will ich mich hier auf einen häufigen und sehr interessanten Fall beschränken. Die Platte befindet sich im Standby Mode und es liegt dem Marktplatz bereits ein Gebot eines anderen Benutzerprozesses vor. Das durch den oben gezeigten read−Aufruf entstehende Gebot reicht aus, die Aktion anzustoßen. Abbildung 21 zeigt die dabei zwischen den Komponenten verschickten Nachrichten.

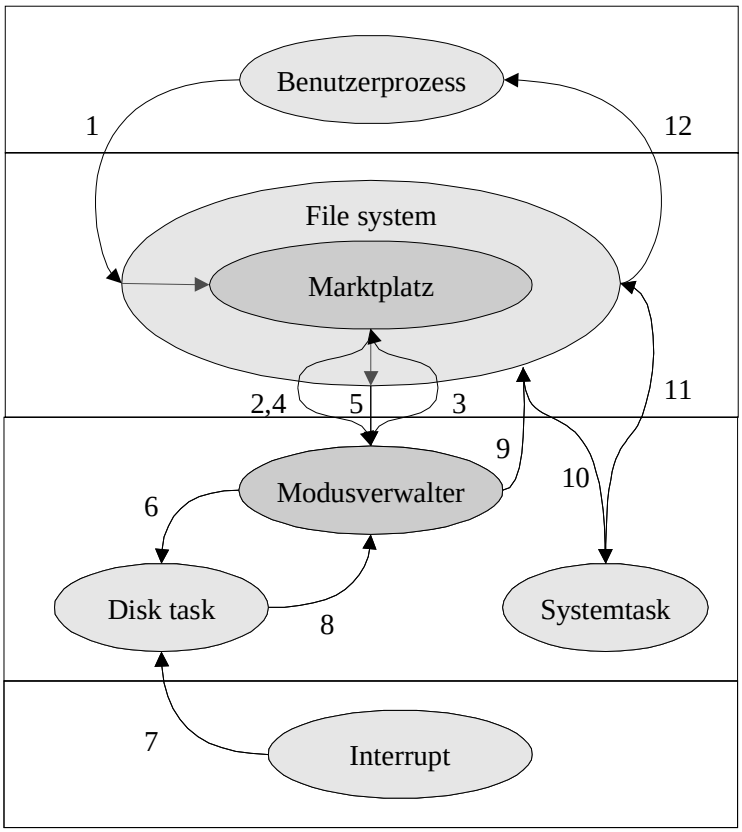

Abbildung 21: Nachrichten beim read−Aufruf

1. Der Benutzerprozess tätigt den read−Aufruf. Dadurch versendet der entsprechende Stub in der C−Bibliothek eine Nachricht an das Filesystem. In dieser Nachricht sind Aufrufkennzahl und Parameter enthalten.

   Das Filesystem empfängt die Nachricht und packt die enthaltenen Daten aus (Demarshalling). Über die Aufrufkennzahl kann die richtige Behandlungsroutine (read) gefunden werden. In dieser Routine wird festgestellt, dass sich der Aufruf auf eine Festplattendatei bezieht. Der Aufruf ist damit prinzipiell kostenpflichtig. Anhand des Dateideskriptors kann erkannt wer−den, dass ein Dateikonto existiert. Die ursprüngliche Nachricht, sowie die Position des Dateikontos werden dem Marktplatz mit übergeben, wenn mittels `check_energy` ein Ge−bot für diesen Aufruf abgegeben wird.

   Der Marktplatz überprüft, ob das Dateikonto genügend Energie aufweist, um ein Gebot von 2 Joule zu decken. Dies ist mit 9,3 Joule der Fall.

2. Daraufhin wird eine Preisanfrage an den Modusverwalter geschickt. Der Modusverwalter weiß, dass sich die Festplatte im Standby Mode befindet. Da das Modusverwalterkonto leer ist, wird der volle Preis für eine Aktion in diesem Modus gefordert. Die Platte ist lange genug im Standby Mode (mehr als 37 Sekunden), so dass nur die Spinup−Kosten anfallen.

3. Der Modusverwalter antwortet dem Marktplatz auf seine Preisanfrage und fordert 8,8 Joule.

   Auf dem Marktplatz wartet bereits ein Gebot von 7,5 Joule. Bei der Aufrechnung der Gebote

(7,5 + 2 Joule) wird festgestellt, dass genug Energie vorhanden ist, um die geforderten Aktionen auszuführen.

Der Marktplatz kassiert deshalb die gebotene Summe. Da nur 8,8 Joule nötig sind, werden vom Dateikonto nur die fehlenden 1,3 Joule abgebucht.

4. Die 8,8 Joule werden vom Marktplatz an den Modusverwalter überwiesen. Dazu wird eine Nachricht vom Typ Transferbuchung an den Modusverwalter geschickt.

   Der Modusverwalter schreibt die 8,8 Joule seinem Modusverwalterkonto gut.

   Der Marktplatz lässt daraufhin den read−Aufruf passieren und übergibt die Kontrolle zurück an die Behandlungsroutine des Filesystems.

   Dort wird es während der Ausführung notwendig, einen Block von der Festplatte zu lesen.

5. Dazu wird eine Nachricht mit dem entsprechenden Befehl an den Modusverwalter verschickt.

   Der Modusverwalter empfängt diese Nachricht. Da auf dem Modusverwalterkonto genügend Energie vorhanden ist, um die Aktion auszuführen, kann fortgefahren werden.

6. Die Nachricht kann damit an die Disk Task weitergeleitet werden. Dieser führt das entsprechende Kommando aus und wartet auf Antwort von der Platte.

7. Der Hardware−Interrupt wird von Schicht 1 in eine Nachricht umgesetzt und an die wartenden Disk−Task verschickt.

8. Diese informiert den Modusverwalter, dass die Aktion ausgeführt wurde.

   Der Modusverwalter registriert, dass die Platte jetzt hochgefahren ist und zieht die entsprechenden Kosten vom Modusverwalterkonto ab.

9. Die Rückantwort trifft beim Filesystem ein. Nun müssen nur noch die Daten in den Benutzeradressraum kopiert werden.

10. Dazu wird eine Nachricht an den System Task verschickt, der dafür zuständig ist.

11. Der System Task quittiert den Erfolg mit einer Rückantwort.

12. Das Filesystem schickt eine Antwort an den Benutzerprozess. Die Ergebnisse werden vom Stub der C−Bibliothek ausgepackt und der aufrufenden Routine übergeben.

4.3. Parametrierung des Systems

Nach der Implementation des Konzepts und dem Einbetten neuer Komponenten ist es wichtig, geeignete Werte für die Parameter des Systems zu finden. Erst dann kann im laufenden Betrieb getestet werden.

Für folgende Parameter sind Werte zu finden:

- Preis im Low Performance Idle Mode
- Preis im Standby Mode
- Länge des Standby Timeouts

4.3.1. Preis im Low Performance Idle Mode

Dieser Preis wird immer dann gefordert, wenn sich die Platte im Low Performance Idle Mode befindet und Schreib− oder Leseanforderungen eintreffen. Wie in Kapitel 2.2.2 erläutert, können die Kosten hier nicht durch einfache Messung der Leistungsaufnahme bestimmt werden. Zu viele nicht bekannte Parameter lassen die Messwerte schwanken. Es soll daher mit einem Mittelwert gearbeitet werden, auch wenn dieser naturgemäß nicht den exakten realen Kosten in jedem Einzelfall entsprechen kann. Es stehen jedoch leider keine Alternativen zur Verfügung.

Bei der Suche nach einem Weg, einen solchen Mittelwert dennoch möglichst realitätsbezogen zu ermitteln, tritt ein Problem auf. Einerseits soll dieser Parameter über den tatsächlichen Energieverbrauch bestimmt werden, andererseits hat er selbst Rückwirkungen darauf. Je höher der Preis im Low Performance Idle Mode ist, desto mehr Gebote sind nötig, bis er bezahlt werden kann und desto mehr Zugriffe auf die Platte erfolgen anschließend kurz hintereinander. Letzteres hat wiederum Auswirkungen auf den Energieverbrauch und damit auf den Preis. Daher waren zwei Schritte notwendig. In einem ersten Schritt wurde aus der Kurve für einen write−Aufruf in Abbildung 6 numerisch die verbrauchte Energie für diesem Fall berechnet. Der ermittelte Wert kann als grobe Schätzung herangezogen werden und kommt in einer ersten Implementierung vorläufig für den hier zu ermittelnden Preis zum Einsatz. Anschließend können am laufenden System nochmals Messungen durchgeführt werden.

Das Ergebnis ist ein Mittelwert von 3,23 Joule, was zwischen den Kosten der Kurven für einen und für 10 write−Aufrufe liegt. Somit beträgt der Preis 32 (in 0,1 Joule).

4.3.2. Preis im Standby Mode

Der Preis im Standby Mode ist erheblich komplexer zu berechnen. Es handelt sich nicht um einen festen Wert, sondern um eine zeitabhängige Formel, nach der der Preis bei jeder Anfrage neu berechnet wird. Um die bereits in Kapitel 3.4.2 gewonnene Formel (Abbildung 11) zu konkretisieren, soll sie hier nochmals aufgegriffen werden.

$$Preis_{Standby\ Mode}(t)=\begin{cases} \widehat{E_{SU}}+C(t_{PD}) & 0\le t<t_{PD} \\ \widehat{E_{SU}}+C(t) & t_{PD}\le t<t_{SB}-t_{SU} \\ \widehat{E_{SU}} & t_{SB}-t_{SU}\le t<\infty \end{cases}$$

Das Zustandekommen der Formel ist bereits erklärt worden. Hier soll sie mit realen Werten angefüllt werden. Dabei kann auf die Ergebnisse der ausführlichen Messungen in Kapitel 2.2.2 zurückgegriffen werden. Alle benötigten Messwerte wurden dort bereits ermittelt. Setzt man diese in die obige Formel ein, erhält man folgende Werte:

$$Preis_{Standby\ Mode}(t) = \left\{ \begin{array}{cc} 23{,}30\,J & 0 \le t < 14\,s \\ 35{,}62\,J - 0{,}88 \cdot t & 14\,s \le t < 30{,}5\,s \\ 8{,}82\,J & 30{,}5\,s \le t < \infty \end{array} \right\}$$

Der Term C(t) wurde ebenfalls bereits mit den realen Werten berechnet und so weit wie möglich vereinfacht. Setzt man den Grenzwert von 30,5 s in die zweite Zeile ein, so fällt eine kleine Sprungstelle am Übergang zum dritten Intervall auf (8,78 J statt 8,82 J). Es handelt sich dabei lediglich um einen Rundungsfehler (der exakte Zeitwert liegt etwas über 30,45 s). Diese Werte müssen aber ohnehin noch überarbeitet werden. Wie in Kapitel 4.2.2 vorgestellt, werden Energiewerte in der Implementation auf 0,1 Joule genau, Zeitwerte sekundengenau gemessen.

Daraus folgt, dass weiter gerundet werden muss. Da damit auch der Faktor 0,88 auf 9/10 steigt, ist der Wert für das erste Intervall (obere Zeile) etwas angehoben, um weiterhin einen weichen Übergang zu erhalten. Dieser Wert ist ohnehin am wenigsten realitätsnah. Es wird aus Mangel an genaueren Messkurven angenommen, dass der Powerdown abgeschlossen ist. Daher kann diese Korrektur vertreten werden, zumal das Intervall auch das kürzeste ist. Auch die Sprungstelle zum dritten Intervall wurde auf diese Weise beseitigt. Die angepasste Formel lautet dann (Energiewerte sind jetzt, wie in der Implementation, in 0,1 J angegeben):

$$Preis_{Standby\ Mode}(t) = \left\{ \begin{array}{cc} 241 & 0 \le t < 14 \\ 367 - 9 \cdot t & 14 \le t < 31 \\ 88 & 31 \le t < \infty \end{array} \right\}$$

Abbildung 22 zeigt den Preis nach dieser Formel graphisch.

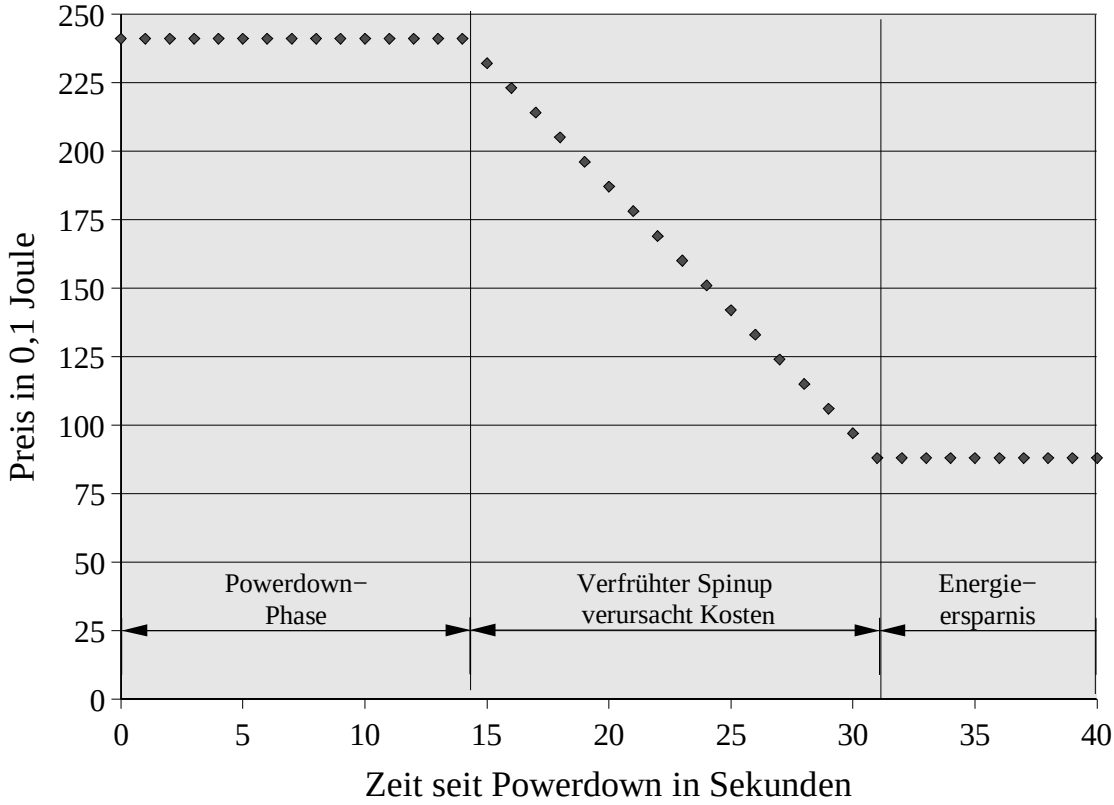

Abbildung 22: Preis im Standby Mode in der Implementierung

4.3.3. Länge des Standby Timeouts

Der Standby Timeout legt fest, wie viele Sekunden nach dem letzten Zugriff auf die Festplatte vom Low Performance Idle Mode in den Standby Mode gewechselt werden soll. Dabei ist es ir−relevant, ob weitere Gebote auf dem Marktplatz vorliegen. Dieser Parameter ist am schwersten zu bestimmen, da hier Prognosen für die Zukunft gestellt werden müssen. Welcher Wert brauchbare Ergebnisse liefert, hängt von der Charakteristik der Zugriffe ab. Untersuchungen in diese Richtung wurden aus Zeitgründen nicht vorgenommen. Kapitel 5 stellt aber einen Denk−ansatz vor, wie mit Hilfe des Marktplatzes solche Analysen zu realisieren wären.

Beim gegenwärtigen Stand der Implementierung ist daher nur ein fester Timeout vorgesehen. Dieser kann im laufenden System über den in Kapitel 4.2.2 vorgestellten powerctl−Aufruf an die aktuelle Situation angepasst werden. Im Testsystem wurde meistens mit einem Timeout von 30 Sekunden gearbeitet, was zu brauchbaren Ergebnissen geführt hat.

4.4. Erstellung von Testanwendungen

Nachdem nun das Betriebssystem soweit vorbereitet worden ist, stellt sich die Frage, wie An−wendungsprogramme die zur Verfügung gestellten Mechanismen optimal nutzen können. Da das vorgestellte Konzept nicht rein betriebssystemseitig arbeitet, sondern gerade so konzipiert ist, dass den Anwendungen eine erhebliche Mitverantwortung bei der Energieeinsparung zuteil wird, soll jetzt die Anwendungsseite beleuchtet werden.

Hierbei gilt es, Strategien vorzustellen, wie Anwendungen ihre Lese− und Schreibzugriffe mög−lichst energiesparend und trotzdem in akzeptabler Zeit abwickeln können. Sicherlich kann es keinen allgemein gültigen Algorithmus geben. Zu unterschiedlich sind die Anforderungen, die in den einzelnen Situationen gestellt werden. Für zwei häufige Fälle sollen hier jedoch Strategien vorgestellt werden.

4.4.1. Dateioperationen ohne strenges Zeitlimit

Viele Dateizugriffe müssen nicht sofort ausgeführt werden. Ein Beispiel dafür ist das verbreitete „Automatische Speichern“. Bekannt vor allem aus Textverarbeitungen, findet es sich heute in einer breiten Palette von Programmen, mit denen zum Teil große Dokumente kontinuierlich überarbeitet werden. Dabei wird eine Zeitspanne (üblicherweise einige Minuten) festgelegt, nach welcher eine neue Version des aktuell bearbeiteten Dokuments auf dem Datenträger gesichert wird. Geht durch einen Programmfehler, einen Systemausfall oder einen Bedienungsfehler die im Hauptspeicher befindliche aktuelle Version verloren oder wird unbrauchbar, so kann auf die Sicherung zurückgegriffen werden. Diese ist maximal die konfigurierte Zeitspanne alt.

Aus energetischer Sicht ist klar, dass ein Verzicht auf dieses periodische Speichern die Funk−tionalität der Anwendung nicht beeinträchtigen, jedoch Energie einsparen würde. Allerdings wäre der Schutz gegen einen möglichen Datenverlust vermindert. Mit Hilfe des Konzepts der kooperativen Nutzung ist leicht ein Kompromiss zu finden. Da es weniger wichtig ist, das au−tomatische Speichern zu einem bestimmten Zeitpunkt auszuführen, kann das Augenmerk darauf gelegt werden, möglichst wenig Energie zu verbrauchen. Durch Begrenzung des Energiever−brauchs für das Schreiben der Daten kann diese Wirkung erzielt werden.

Dabei bieten sich mehrere Möglichkeiten an, wie das konkret zu realisieren ist. Sie sind abhängig davon für, wie wichtig das regelmäßige Schreiben von Sicherungskopien beurteilt wird.

Variante 1

Sicherheitskopien sollen angelegt werden, wenn sich eine − energetisch günstige − Gelegenheit dazu bietet. Auf keinen Fall soll zusätzliche Energie verbraucht werden.

Beispiel: Der update−Daemon

MINIX umfasst von Haus aus ein Systemprogramm, das zwar keine Dokumente speichert, dessen Festplattenzugriffe aber genau in diese Problemklasse fallen: *update*. Dabei handelt es sich um einen im Hintergrund laufenden Prozess, der alle 30 Sekunden einen sync−Systemaufruf absetzt, um geänderte Blöcke im Puffer des Filesystems auf die Platte zu schreiben. Ziel ist es, das System gegenüber Ausfällen oder versehentlichem Abschalten resistenter zu machen. Der update−Daemon ist daher nicht nur eine beispielhafte Testanwendung, er sollte sogar dringend in das Konzept einbezogen werden, da ein sync−Aufruf im 30−Sekunden−Takt alle Bemühungen, die Platte möglichst energiesparend zu betreiben, zunichte macht. Die POSIX−Variante gibt auf dem Marktplatz stets ein Gebot in der Höhe des Prozesskontos ab. Ein derart hohes Gebot führt selbstverständlich immer zur sofortigen Ausführung.

Abbildung 23 zeigt den geänderten Quellcode des update−Daemons. Die einzige Änderung be−steht darin, dass der sync−Aufruf durch einen sync2−Aufruf ersetzt wird.

```
/* update − do sync periodically          Author:   Andrew S. Tanenbaum
 *                                        Modified: Steffen Meyer
 */

#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

_PROTOTYPE(int main, (void));

int main()
{

  /* Release all (?) open file descriptors. */
  close(0);
  close(1);
  close(2);

  /* Release current directory to avoid locking current device. */
  chdir("/");

  /* Flush the cache every 30 seconds. */
  while(1) {
    /* sync for free, no matter how long it takes */
    sync2(0, −1);

    sleep(30);
  }

}
```

Abbildung 23: Implementation des update−Daemons (/usr/src/commands/simple/update.c)

Es kommt die in Kapitel 4.2.2 vorgestellte Variante sync2 des Aufrufs zum Einsatz, die eine Beschränkung des Energieverbrauchs zulässt. Die gebotene Energie beträgt immer 0 Joule bei deaktiviertem Timeout. Dieses Vorgehen hat den Effekt, dass *update* nie selbst Schreibzugriffe auslöst, da das Energiegebot dazu nicht ausreicht. Stattdessen wird der interne Puffer des Filesystems immer im Zuge anderer Plattenaktivitäten geleert. Ein solches synchronisiertes Schreiben bedeutet, wie bereits am Ende von Kapitel 2.2.2 erläutert, dass der energetische Mehraufwand sehr gering ausfällt.

Problematisch wird diese Variante, wenn andere Prozesse längere Zeit keine Plattenzugriffe tä−tigen, in deren energetischen „Windschatten“ der update−Daemon schreiben kann. Dann bleibt auch das regelmäßige Leeren der Puffer aus. Das gezeigte Vorgehen eignet sich in diesem Fall trotzdem sehr gut. Der Filesystem−Puffer füllt sich nur, wenn Dateizugriffe stattfinden. Daher ist ein Leeren der Puffer auch nur notwendig, wenn Dateizugriffe stattfinden.

Variante 2

Bei dieser Variante sollen ebenfalls Sicherungskopien angelegt werden, wenn sich eine energe−tisch günstige Gelegenheit dazu bietet. Allerdings besteht die Zusatzforderung, dass ein zeitliches Limit nicht überschritten werden darf, nachdem auf jeden Fall eine Kopie geschrieben sein muss. Diese Variante soll verhindern, dass beispielsweise eine Textverarbeitungen, die als einzige Anwendung gestartet ist, aufgrund des eben genannten Problems gar keine oder nur in inakzeptablen Abständen Sicherheitskopien anlegt.

Das Idee ist dabei folgende. Der Benutzer gibt zwei Zeiten vor: eine minimale und eine maxi−male Zeitspanne für das Anlegen der Sicherungskopien. Das Programm versucht, diese

Forderung mit minimalem Energieaufwand zu erfüllen. Nach dem Verstreichen der minimalen Sicherungszeit werden Gebote mit mittlerem, festem Timeout abgegeben, die in ihrer Höhe linear mit der verstrichenen Zeit ansteigen. Droht ein Erreichen der maximalen Sicherungszeit, so wird zusätzlich der Timeout kontinuierlich erniedrigt. Dadurch ergibt sich ein exponentieller Anstieg der Gebote, der zum Zeitpunkt der maximalen Zeit (theoretisch) gegen unendlich geht.

Abbildung 24 zeigt eine Testimplementierung dieses Algorithmus. Das Programm ist der Übersichtlichkeit wegen auf den wesentlichen Teil reduziert. Öffnen und Schließen der Datei, auf die geschrieben wird, sind nur als Kommentare angedeutet. Pro Durchlauf der äußeren Schleife wird eine Version der Datei geschrieben. Die innere Schleife dient dazu, bei gescheitertem write-Aufruf das Gebot entsprechend zu steigern. Konfiguriert ist eine minimale Zeit (`MIN_TIME`) von 3 Minuten, eine maximale Zeit (`MAX_TIME`) von 8 Minuten und ein Start-Timeout (`TIMEOUT`) von 10 Sekunden. Aus diesen Werten wird berechnet, wann der exponentielle Anstieg einsetzten muss (Variable `critical`). Das Gebot wird stets um 5 (= 0,5 Joule) erhöht (`ENERGY_STEP`).

Im exponentiellen Teil wird der Timeout jeweils um eine Sekunde verkürzt. Daher lässt sich `critical` leicht berechnen, indem von der maximalen Zeit die Zeit abgezogen wird, die vergeht, wenn der momentane Timeout in 1 Sekunden-Schritten erniedrigt wird. Auf diese Weise wird erreicht, dass die Gebote exakt bei `MAX_TIME` gegen unendlich gehen. Dabei wurde von der bekannten Gaußschen Formel für die Berechnung der Summe 1+2+3+4+.... Gebrauch gemacht. Formal berechnet sich die kritische Zeit durch:

$$t_{critical} = t_{max} - \sum_{i=1}^{t} i$$

Bei der Wahl der Parameter ist darauf zu achten, dass die kritische Zeit nicht kleiner als die minimale Zeit sein darf, da sonst die maximale Zeit überschritten wird. Ist der Timeout auf Null gesunken (beim Erreichen der maximalen Zeit), wird er deaktiviert und alle zur Verfügung stehende Energie geboten. Da der Prozess nie mehr als das momentane Energiedefizit auf dem Marktplatz zahlen muss, ist dies die beste Lösung (Diese Problematik wird Kapitel 4.4.2 genauer dargelegt).

```
/* autosave − simulate energy sensitive autosaving
 * Author:   Steffen Meyer
 */

#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>

#define MIN_TIME 180
#define MAX_TIME 480
#define TIMEOUT   10
#define ENERGY_STEP 5
#define NR_OF_ITERATIONS 10

int main(int argc, char** argv)
{
  int fd, i;

  for(i=0; i<NR_OR_ITERATIONS; i++) {
    int energy=0, timeout=TIMEOUT, time_since_last_autosave = MIN_TIME;

    /* ... open file ...*/
    while(1) {
      int ret, critical;

      if(timeout==0) energy = timeout = −1;
      ret = write2(fd, "Teststring", 10, energy, timeout);
      if((ret==−1) && (errno==ENOENERGY)) {
            /* energy was not enough */

            energy += ENERGY_STEP;
            time_since_last_autosave += timeout;

            critical = MAX_TIME −((timeout*(timeout+1)/2));
            if(time_since_last_autosave > critical) {
              /* exponetial rise */
              timeout−−;
            }
            continue;
      }
      else break;
    }
    /* .... close file ... */
    sleep(MIN_TIME);
}}
```

Abbildung 24: Testimplementierung des Autosave−Algorithmus

Der anfängliche Timeout hat dabei große Auswirkung auf das Gesamtverhalten des Algorithmus. Er bestimmt nicht nur, in welchem Abstand die Gebote beim Start erfolgen. Eng daran ist auch geknüpft, wann die exponentielle Phase beginnt und wie schnell die Gebote im zeitlichen Ver−lauf steigen. Abbildung 25 zeigt die Entwicklung der Gebote bei Start−Timeout 10 (links) und Start−Timeout 15 (rechts). Als minimale Zeit wurden 3 Minuten, als maximale Zeit 8 Minuten gewählt. Jedes neue Gebot liegt um 0,5 Joule über dem vorherigen (konstante Steigerung). Es ist leicht zu sehen, dass ein niedriger Start−Timeout zu einem steileren Anstieg der Kurve und spä−teren Beginn der exponentiellen Phase führt. Der steilere Anstieg könnte jedoch durch einen kleineren Wert für *ENERGY_STEP* abgefangen werden. Den Kurven gemeinsam ist der steile Anstieg im Bereich des maximalen Wertes (8 Minuten), der ja gewollt ist. Nur so kann sicher−gestellt werden, dass die Maximalzeit nicht überschritten wird.

Allgemein ist zu erkennen, dass durch Variieren der vier Parameter *MIN_TIME*, *MAX_TIME*, *TIMEOUT* und *ENERGY_STEP* leicht für nahezu beliebige Fälle ein passender Algorithmus ge−funden werden kann. Dies gilt nicht nur für den Bereich des automatischen Speicherns, sondern für viele Situationen, in denen eine Verzögerung der Ausführung toleriert werden kann.

Natürlich besteht ein Speichervorgang in der Regel nicht aus einem einzigen write−Aufruf. Auch das Öffnen einer Datei ist mit Kosten verbunden. In der Praxis wird es daher nötig sein, diesen Algorithmus mehrmals auszuführen. Dabei muss die Anwendung natürlich darauf achten, dass der gegebene Zeitrahmen für die Summe der write−Aufrufe nicht überschritten wird und die Maximalzeiten für die Einzelaufrufe entsprechend klein gewählt werden. In den für die Mes−sungen verwendeten Testanwendungen wurde auf solche Fälle bewusst verzichtet, um die Beispiele verständlich zu halten. Die Wirkungsweise des Konzepts kann der vorgestellte, einfa−che Algorithmus besser demonstrieren.

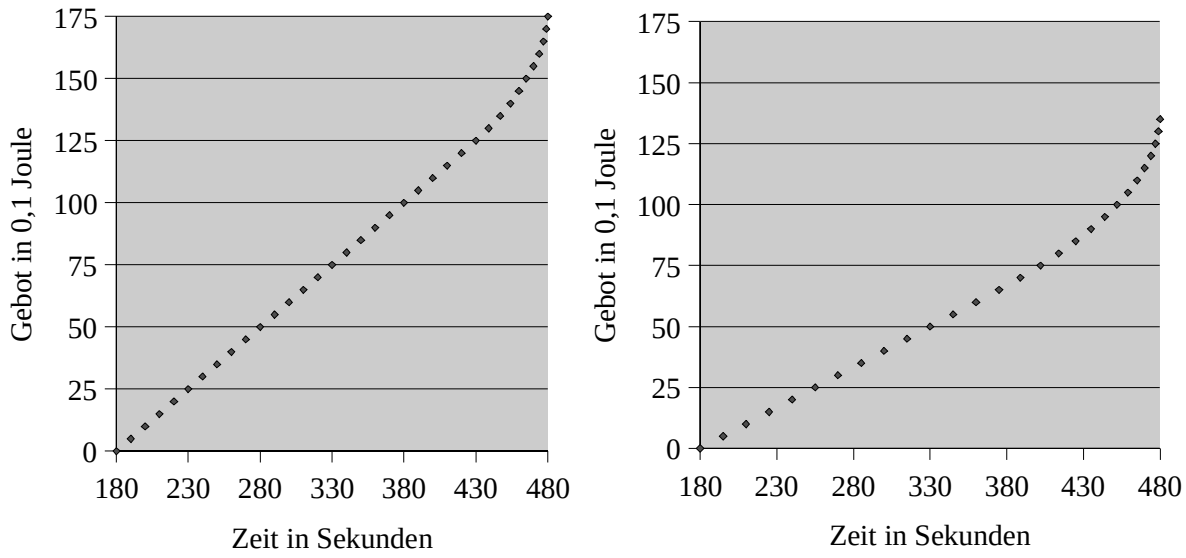

Abbildung 25: Abgegebene Gebote beim Autosave−Algorithmus (Start−Timeout 10 und 15)

4.4.2. Zeitkritische Dateioperationen

Ein anderer Problembereich sind zeitkritische Operationen. Dazu gehören nicht nur Echtzeit−anwendungen, sondern alle Fälle, bei in denen eine Verzögerung den laufenden Betrieb maßgeblich behindert. Kein Benutzer nimmt beim Öffnen eines Dokuments in einer Textverar−beitungen aus Energiespargründen Verzögerungen im Minutenbereich hin. Hier sind die Prioritäten genau umgekehrt im Vergleich zum automatischen Speichern: Die Verzögerung muss sehr klein sein, trotzdem soll, wenn möglich, nicht viel Energie aufgewendet werden.

Die Problematik ist offensichtlich: Einerseits muss schnell eine Einigung auf dem Marktplatz erzielt werden, was die Anzahl der möglichen Gebote bis zum Erfolg begrenzt. Auf der anderen Seite ist mit wenigen Geboten ein „Herantasten“ an den einen optimalen Preis nicht möglich. Im konkreten Fall macht sich hier die kleinste Zeiteinheit von einer Sekunde bemerkbar. Für zeit−kritische Anwendungen kann ein Timeout von einer Sekunde bereits zu hoch sein oder zumindest die Anzahl der Versuche auf sehr wenige (2−3) begrenzen. Ein Bieten bei Timeout Null ist nur begrenzt sinnvoll. Der Modusverwalter berechnet den Preis nur im Sekundentakt neu. Die Situation auf dem Marktplatz kann sich also zwischenzeitlich nur durch Gebote anderer Prozesse ändern. Die hohe Frequenz der Gebote bei Timeout Null lässt dies unwahrscheinlich werden, zumal diese „Überflutung“ mit Anfragen den Marktplatz und das ganze Filesystem

schnell überlasten würde. In diesem Fall sollte gleich ein sehr hoher Wert geboten werden, denn mehr als das aktuell auf dem Markt bestehende Energiedefizit muss der Prozess niemals zahlen.

Fazit dieser Überlegungen: Ein Prozess, der in weniger als einer Sekunde einen Dateizugriff ausführen muss (Echtzeitanwendung), kann in der aktuellen Implementierung selbst keine Energie sparen. Er sollte stets den Wert seines Prozesskontos als Gebot abgeben. Dazu kann das POSIX−Interface der gängigen Aufrufe verwendet werden. Jedoch bietet sich anderen Prozessen dadurch die Möglichkeit, günstige Dateizugriffe zu tätigen, so dass global gesehen trotzdem Energie gespart wird.

Um bei Messungen ein möglichst realistisches Szenario aufbauen zu können, ist eine Testanwendung nötig, die zeitkritische Dateioperationen tätigt. Die Zugriffe auf die Festplatte sollen nicht in regelmäßigen Abständen erfolgen, sondern das eher zufällig verteilte Laden und Speichern von Dokumenten durch einen Benutzer simulieren. Daher bietet sich auf den ersten Blick ein Algorithmus an, bei dem der zeitliche Abstand zwischen den Zugriffen durch eine Zufallszahl bestimmt wird. Dies hat aber den Nachteil, dass die Ergebnisse der Messungen nicht rekonstruierbar sind. Folglich muss ein anderer Weg gewählt werden.

Eine Lösung besteht darin, statt einer (meist zeitabhängigen berechneten) Zufallszahl eine Pseudozufallszahl zu verwenden. Pseudozufallszahlen werden häufig durch ein iteratives Verfahren berechnet. Bei gleichen Parametern und Startwerten resultiert daraus immer die gleiche Zahlenfolge. Ein einfach zu implementierender Algorithmus ist der lineare Kongruenz−Algorithmus, wie er in [KNU98] beschrieben ist. Er bildet die Basis für Berechnung der zeitlichen Abstände der Zugriffe in der Testimplementierung. Die verwendeten Parameter wurden unter Berücksichtigung der in [KNU98] genannten Rahmenbedingungen frei gewählt. Der gelieferte Wert liegt zwischen 0 und 1 und wird auf ein konfigurierbares Zeitintervall (*MIN_TIME* ... *MAX_TIME*) hochgerechnet. Abbildung 26 zeigt die Testimplementierung.

```
/* urgentsave − simulate time critical saving
 * Author:   Steffen Meyer
 */

#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>

#define MIN_TIME 180
#define MAX_TIME 480
#define RANDOM_FIRST 1231
#define NR_OF_ITERATIONS 50

_PROTOTYPE(int main, (void));
_PROTOTYPE(int Random, (void));

int main(int argc, char** argv)
{
  int fd, i;

  /* .... open file ....*/

  for(i=0; i<NR_OF_ITERATIONS; i++) {
    write(fd, "Teststring", 10);
    sleep(Random());
  }
  /* .... close file ...*/
}

/* calculate "random" time */

int Random() {
  static int S = RANDOM_FIRST;
  int A = 20001, B = 3, M = 1048576;

  S = (A*S+B)%M;
  if(S<0) S = −S;

  return(S*(MAX_TIME−MIN_TIME)/M+MIN_TIME);
}
```

Abbildung 26: Test−Implementierung des urgentsave−Algorithmus

Das Öffnen und Schließen der Datei wurde aus Übersichtlichkeitsgründen wieder nur angedeutet. Die Parameter `MIN_TIME` und `MAX_TIME` bestimmen den Bereich der Verzögerung zwischen zwei Zugriffen, `RANDOM_FIRST` gibt einen Startwert für den Zufallszahlenalgorithmus vor; `NR_OF_ITERATIONS` legt die Anzahl der auszuführenden Zugriffe fest. Solange `RANDOM_FIRST` unverändert bleibt, wird immer die gleiche Folge von „Zufallszahlen“ durchlaufen.

4.4.3. Testsets

Für Praxistests stehen drei Programme zur Verfügung: *update*, *autosave* und *urgentsave*. Der update−Daemon wird bereits beim Hochlaufen des Systems gestartet, da er für ein stabiles Filesystem wichtig ist. a*utosave* und *urgentsave* stehen in der in Abbildung 24 und 26 gezeigten Form zur Verfügung, allerdings wurden die Programme soweit verändert, dass die am Programmanfang fest kodierten Parameter stattdessen auf der Kommandozeile übergeben werden können.

Die Implementierung von *autosave* umfasst zusätzlich das in Abbildung 24 nur angedeutete Öffnen und Schließen der Datei. Da das Öffnen ebenfalls Kosten verursacht, kommt hier der gleiche Algorithmus wie beim Schreiben selbst zum Einsatz. Es wird also zweimal der Biete−prozess durchlaufen, einmal für das Öffnen, ein zweites Mal für das Schreiben der Datei. Die minimale Zeit wird nur einmal gewartet, nachdem die Datei wieder geschlossen wurde. Die maximale Zeit dagegen kommt sowohl beim Öffnen, als auch beim Schreiben zum tragen. Damit kann ein Durchlauf (Öffnen, Schreiben, Schließen) bis zu zweimal `MAX_TIME` dauern.

Aus verschiedenen Konfigurationen von *autosave* und *urgentsave* wurden zwei Testsets ge−schaffen. Um das Anstossen der Tests zu erleichtern, existieren Shell−Skripte, die die benötigten Instanzen mit den zugehörigen Parametern parallel starten. Die Testsets sollten praxisnah sein und die Möglichkeiten des Konzepts aufzeigen. Daher wurden mehrere Instanzen von *autosave* und *urgentsave* mit unterschiedlichen Parametern aufgenommen. *urgentsave* muss nur einmal vertreten sein, denn die Höhe der Gebote ändert sich bei diesem Algorithmus nicht (es wird immer der Maximalwert geboten). Daher können mehrere Anwendungen leicht durch niedrigere Zeitparameter `MIN_TIME` und `MAX_TIME` simuliert werden.

Die Tabellen 6 und 7 geben einen Überblick über die verwendeten Parameter. Das Vorgehen bei den Praxistests und die gewonnenen Erkenntnisse werden im nachfolgenden Kapitel dargelegt.

Testset A: 4x autosave, 1x urgentsave				
autosave:				
MIN_TIME	***MAX_TIME***	***TIMEOUT***	***ENERGY_STEP***	***NR_OF_ITERATIONS***
60	180	5	5	4
90	180	10	5	3
120	240	10	7	3
120	300	15	7	3
urgentsave:				
MIN_TIME	***MAX_TIME***	***RANDOM_FIRST***	***NR_OF_ITERATIONS***	
120	240	1231	2	

Tabelle 6: Aufbau und Parameter von Testset A

Testset B: 6x autosave, 1x urgentsave				
autosave:				
MIN_TIME	***MAX_TIME***	***TIMEOUT***	***ENERGY_STEP***	***NR_OF_ITERATIONS***
30	120	5	5	5
60	120	10	5	5
120	240	10	7	5
120	300	15	7	4
180	300	15	10	4
180	360	18	10	3

Testset B: 6x autosave, 1x urgentsave			
urgentsave:			
MIN_TIME	***MAX_TIME***	***RANDOM_FIRST***	***NR_OF_ITERATIONS***
90	180	1231	4

Tabelle 7: Aufbau und Parameter von Testset B

4.5. Verhalten im laufenden Betrieb

Ein Energiesparkonzept, das in der Theorie gut funktioniert, muss nicht zwangsläufig in der Praxis gute Ergebnisse liefern. Daher ist es unabdingbar, ausführliche Tests durchzuführen.

4.5.1. Kooperatives und konventionelles Konzept im Vergleich

Ein wichtiges Kriterium für die Bemessung des kooperativen Konzepts ist der Vergleich mit einem Referenzsystem, das eine konventionelle Energiesparstrategie verfolgt. Hierfür würde sich auf den ersten Blick ein unmodifiziertes Standard−MINIX anbieten. Da MINIX aber von Haus aus keine Energiesparmechanismen besitzt, müsste das konventionelle Konzept erst integriert werden. Es sieht einen Timeout vor, nach dem die Platte in den Standby−Mode wechselt. Der nächste Zugriff auf die Festplatte bewirkt einen erneuten Spinup. Dieses Verhalten kann jedoch vom neu geschaffenen System simuliert werden, indem die Testanwendungen stets sehr hohe Gebote abgeben (in der Höhe ihres Prozesskontos). Dadurch ist ein sofortiger Spinup gewähr−leistet. Ein Vorteil dieser Methode liegt darin, dass die implementierten Mechanismen zur Bemessung der verbrauchten Energie auch im Referenzsystem zur Verfügung stehen.

Analysen von Probeläufen mit dem im letzten Kapitel vorgestellten Testset A haben ergeben, dass die Laufzeiten im neuen System erwartungsgemäß erheblich länger sind, als in einem kon−ventionellen System. Das Platzieren von Geboten und Warten auf Mitinteressenten verzögert die Ausführung der Programme. Ein direkter Vergleich der Gesamtenergieaufnahme ist daher nicht sinnvoll, da eine höhere Laufzeit einen größeren Basisenergieverbrauch für die Laufzeit des Tests bedeutet.

Ein aussagekräftigeres Kriterium in diesem Vergleich ist der aktionsbezogene Verbrauch. Damit ist der zusätzliche Verbrauch gemeint, den die Prozesse durch ihre Aktionen verursachen. Dieser wird vom Betriebssystem bereits ermittelt. Es müssen lediglich die Testapplikationen soweit verändert werden, dass am Beginn und Ende der Stand des Prozesskontos abgefragt wird. Die Differenz dieser Werte gibt den Verbrauch an, der Gesamtverbrauch aller Applikationen im Testset dient als Vergleichskriterium. Ein berechtigter Einwand gegen dieses Vorgehen ist, dass der niedrigere Basisenergieverbrauch im Standby Mode nur über die Preispolitik des Modus−verwalters eingeht (siehe Kapitel 3.4.2). Da eine Energieersparnis (nach mehr als 31 Sekunden im Standby Mode) nicht über die Preise weitergegeben wird, ergibt die reine Betrachtung des aktionsbezogenen Verbrauchs einen zu hohen Wert. Mit anderen Worten: Das Kriterium be−nachteiligt eine Strategie, die die Platte lange im Standby Mode hält. In der Praxis wird dadurch das kooperative Konzept benachteiligt. Beim Durchlaufen der beiden Testsets zeigt sich, dass das konventionelle Konzept im Durchschnitt früher zu einem Spinup führt. Das kooperative Konzept verlangt, dass genügend Energie auf dem Marktplatz geboten wird. Da ein einziges Gebot in der Regel nicht ausreicht, vergeht Zeit, bis Mitinteressenten gefunden sind.

Beide Testsets wurden bei einem Standby−Timeout von 20 Sekunden nacheinander unter den Bedingungen eines konventionellen Energiesparkonzepts gestartet. Anschließend kamen sie im System mit dem kooperativen Konzept zum Einsatz. Die Tabellen 8 und 9 zeigen die Ergebnisse dieser Tests.

Es handelt sich dabei, wie oben dargelegt, nicht um Messwerte, sondern um den vom Betriebs−system berechneten und über den Marktplatz eingeforderten Energieverbrauch. Gleichzeitig wurden jedoch Messungen des realen Energieverbrauchs angestellt, um zu verifizieren, dass die erhaltenen Werte korrekt und aussagekräftig sind. Die Ergebnisse dazu sind in Kapitel 4.5.3 dargelegt.

Testset A	***Kooperatives Konzept***		***Konventionelles Konzept***	
Komponente des Testsets	***Aktionsbezogener Energieverbrauch in 0,1 Joule***	***Laufzeit in Sekunden***	***Aktionsbezogener Energieverbrauch in 0,1 Joule***	***Laufzeit in Sekunden***
autosave 1	546	549	439	247
autosave 2	167	420	495	277
autosave 3	252	619	343	369
autosave 4	123	450	496	369
urgentsave	265	331	206	333
Summe	1353	2369	1979	1595

Tabelle 8: Vergleich von kooperativen und konventionellem Konzept: Testset A

Wie zu erwarten war, sind die Laufzeiten beim kooperativen Konzept deutlich höher. Die Summe der Einzelzeiten liegt fast 50% über dem Wert im Referenzsystem. Sie hat allerdings nur statistische Bedeutung, da die fünf Programme parallel ausgeführt werden. Ein Vergleich der Maximalwerte ergibt einen Anstieg um etwa 2/3. In der Praxis bedeutet dies aber keinen we−sentlichen Nachteil. Zeitkritische Operationen werden nicht verzögert, was der Wert von *urgentsave* zeigt. Interessant ist, dass die Laufzeiten beim konventionellen Konzept etwas über den erwarteten liegen. Da keine Verzögerung auftreten kann, müsste beim Autosave−Algorith−mus die Laufzeit gleich der n−fachen Minimalzeit sein (n sei die Zahl der Durchläufe). Die Zeiten liegen jedoch um bis zu 9 Sekunden höher. Es handelt sich dabei um Anlaufzeiten der Festplatte (beim Spinup) und kleinen Verzögerungen aufgrund des etwas trägen Schedulings.

Ein Blick auf die Energiewerte macht deutlich, dass mit der längeren Laufzeiten eine beachtliche Energieeinsparung erkauft wurde. Die Energieaufnahme beim kooperativen Konzept liegt für das Testset A fast 1/3 unter der im Referenzsystem. Dieses Ergebnis ist hauptsächlich auf die Syn−chronisation der Zugriffe beim kooperativen Konzept zurückzuführen. Testset A enthält 30 Festplattenzugriffe. Es wird jedoch nur 13 mal eine Einigung auf dem Marktplatz erzielt. Dies bedeutet, dass durchschnittlich zwei bis drei Prozesse an einer Einigung beteiligt waren. Folglich müssten die Kosten auf weniger als die Hälfte sinken. Dies geschieht jedoch nicht, da *urgentsave* einmal einen teuren, frühen Spinup auslöst. Zusätzlich kommt die bereits erläuterte Benachtei−ligung des kooperativen Konzepts zum tragen, da zweimal viel Zeit im Standby Mode verbracht wird.

Testset B	*Kooperatives Konzept*		*Konventionelles Konzept*	
Komponente des Testsets	*Aktionsbezogener Energieverbrauch in 0,1 Joule*	*Laufzeit in Sekunden*	*Aktionsbezogener Energieverbrauch in 0,1 Joule*	*Laufzeit in Sekunden*
autosave 1	369	399	895	157
autosave 2	290	519	688	307
autosave 3	473	990	641	609
autosave 4	241	720	592	489
autosave 5	315	988	494	728
autosave 6	128	640	388	549
urgentsave	439	561	335	559
Summe	2255	4817	4033	3398

Tabelle 9: Vergleich von kooperativen und konventionellem Konzept: Testset B

Ein Vergleich der Laufzeiten zeigt, dass das kooperative Konzept hier besser abschneidet als in Testset A. Die Laufzeiten liegen in der Summe 42% höher als beim konventionellen Konzept. Der Zeit für den Durchlauf des Testsets ist sogar nur 36% größer. Da zwei Prozesse mehr vertreten sind, die Gebote abgeben, kommt die geforderte Energie auf dem Markplatz schneller zusammen. Experimente mit deutlich größeren Testsets haben gezeigt, dass die Laufzeiten mit steigender Zahl der Prozesse weiter sinken.

Die Einsparung beim aktionsbezogenen Energieverbrauch fällt deutlich aus. Der Wert beim kooperativen Konzept liegt 44% niedriger als beim Referenzsystem. Testset B enthält 60 Zugriffe auf die Festplatte. Davon werden im neuen System durch synchrones Schreiben nur 23 berechnet. Zusätzlich löst *autosave 1* beim konventionellen Konzept mehrfach verfrühte Spinups aus (daher der sehr hohe Energiewert). Die Einsparung könnte also durchaus noch höher liegen. Hier stellt sich ein Nebeneffekt des synchronen Schreibens ein. Die Platte wird häufiger in den Standby Mode gewechselt, da die Zugriffe weniger gut über die Zeit verteilt erfolgen. Dies spart zwar global Energie, allerdings wirkt sich das nur auf das Betriebssystemkonto aus, das hier nicht betrachtet wird.

4.5.2. Auswirkung unterschiedlicher Parameterkombinationen

In mehreren Tests sollen die Auswirkungen der vier Parameter des Autosave−Algorithmus (`MIN_TIME`, `MAX_TIME`, `TIMEOUT` und `ENERGY_STEP`) und des Standby Timeouts auf den Energieverbrauch und die verstrichene Zeit untersucht werden. Die Testsets bestehen aus jeweils vier Instanzen der Autosave−Testimplementierung. Dabei wurden stets einige Parameter fest gehalten, während die anderen variieren. Der Betrachtungszeitraum liegt bei fünf Schleifendurchläufen (`NR_OF_ITERATIONS=5`).

Variation des Parameters `MIN_TIME`:

Eine Variation des Parameters *`MIN_TIME`* macht nur Sinn, wenn *`MAX_TIME`* in gleichem Maße verändert wird. Anderenfalls würde sich der Verlauf der Gebotskurve ändern, da die kritische Zeit durch die Differenz von *`MAX_TIME und MIN_TIME`* beeinflusst wird.

Feste Parameter:

TIMEOUT	*ENERGY*	*Standby Timeout*
10	5	20

Testergebnis:

Instanz	*MIN_TIME*	*MAX_TIME*	*Energieverbrauch*	*Laufzeit*
1	40	280	405	689
2	50	290	380	699
3	60	300	305	709
4	70	310	345	889

Das Ergebnis weicht auf den ersten Blick vom erwarteten ab. Von Instanz 1 bis Instanz 3 fällt der Energieverbrauch. Die immer kürzere minimale Zeit bewirkt eine kürzere Wartedauer zwischen den Durchläufen. Daher beginnt Instanz 2 zu einem Zeitpunkt mit den Geboten für den nächsten Durchlauf, an dem Instanz 1 bereits ein höheres Gebot abgibt. Gleiches gilt für Instanz 3. Der niedrigere Energieverbrauch von Instanz 2 und 3 basiert auf der Tatsache, dass die Instanzen gemeinsam einen Preis aushandeln, wobei Instanz 1 mit dem Bieten beginnt, dann steigt Instanz 2 ein, schließlich Instanz 3. Wenn die Einigung erfolgt, hat Instanz 1 bereits ein höheres Gebot als Instanz 2 abgegeben, Instanz 2 ein höheres als Instanz 1.

Um die für eine Einigung notwendige Energie zu erreichen, müssen die Gebote mehrerer Instanzen zusammentreffen. Daher werden Instanz 1 und 2 jeweils auf dem Markplatz blockiert, bis Instanz 3 mit dem Bieten beginnt. Auf diese Weise kommen die ähnlichen Laufzeiten zustande, die jeweils 10 Sekunden Differenz resultieren aus der unterschiedlich langen Wartezeit nach dem fünften Durchlauf. Instanz 1 bis 3 werden offensichtlich durch den Marktplatzes synchronisiert.

Die Werte von Instanz 4 fallen offensichtlich aus der Reihe. Eine genaue Analyse hat gezeigt, dass die „Abnormalität“ nach Durchlauf drei auftritt. Die hohe minimale Zeit und eine kurze Verzögerung des Prozesswechsels durch den Scheduler lassen Instanz 4 zu einem Zeitpunkt mit dem Bieten für den vierten Durchlauf beginnen, als Instanz 1 bis 3 mit ihren Geboten bereits eine Einigung erzielt haben. Instanz 4 ist kurzzeitig alleiniger Bieter, was sowohl eine weitere Verzögerung, als auch höhere Gebote nach sich zieht. Dann beginnen nacheinander Instanz 1, 2 und 3 mit dem Bieten für ihren fünften Durchlauf. Instanz 4 liegt jetzt einen Durchlauf zurück und muss ihren fünften Durchlauf komplett allein bezahlen. Dies bedeutet eine hohen Energie− und Zeitaufwand.

Fazit: Das Ändern von *`MIN_TIME`* zeigt deutlich, wie stark das Konzept die Synchronisation von Festplattenzugriffen fördert, und damit zur Energieeinsparung beiträgt.

Variation des Parameters `MAX_TIME`:

Feste Parameter:

MIN_TIME	*TIMEOUT*	*ENERGY_STEP*	*Standby Timeout*
30	5	5	20

Testergebnis:

Instanz	*MAX_TIME*	*Energieverbrauch*	*Laufzeit*
1	30	305	289
2	60	296	289
3	90	291	289
4	120	273	289

Wie zu erwarten ist, sinkt der Energieverbrauch von Instanz 1 zu Instanz 4 hin. Eine kürzere maximale Zeit bewirkt, dass die Gebote schneller verfallen und damit in ihrer Höhe rascher steigen. Die Laufzeit ist konstant, da alle Instanzen synchron mit dem Bieten beginnen und nach erfolgter Einigung auf dem Marktplatz die gleiche Zeit warten, bis erneut Gebote abgegeben werden.

Variation der Parameters `TIMEOUT` und `ENERGY_STEP`:

Tests bestätigen folgende Vermutungen:

- Ein hoher Timeout führt zu einem niedrigen Energieverbrauch, denn in diesem Fall besteht das Gebot lange auf dem Marktplatz. Zusätzlich erhöht sich das Gebot langsamer. Dadurch steigt die Chance, mit einem niedrigen Gebot beim Aushandeln des Preises Erfolg zu haben.
- Ein hoher Wert für *`ENERGY_STEP`* führt zu einem hohen Energieverbrauch, denn in diesem Fall erhöht sich das Gebot schneller.

Es stellt sich daher die Frage, welche Ergebnisse zu erwarten sind, wenn *`ENERGY_STEP`* im gleichen Maße erhöht wird, wie der Timeout.

Feste Parameter:

MIN_TIME	*MAX_TIME*	*Standby Timeout*
60	120	20

Testergebnis:

Instanz	*TIMEOUT*	*ENERGY_STEP*	*Energieverbrauch*	*Laufzeit*
1	2	2	260	440
2	5	5	288	440

Instanz	*TIMEOUT*	*ENERGY_STEP*	*Energieverbrauch*	*Laufzeit*
3	7	7	281	440
4	10	10	290	440

Es ergibt sich kein einheitliches Bild. Ein Testset mit Werten von 2, 5 und 10 für beide Parameter liefert identischen Energieverbrauch und gleiche Laufzeit in allen drei Instanzen. Daher liegt der Schluss nahe, dass die unterschiedlichen Werte für den Energieverbrauch daher rühren, dass mit der 7 ein Parameter im Set ist, der „nicht zu den anderen passt“. Die für die Gebote wichtigen Parameter Preis im Low Performance Idle Mode (32), Standby Timeout (20), Timeout (2, 5, 10) und *ENERGY_STEP* (2, 5, 10) haben gemeinsame Teiler und sind dadurch mathematisch verwandt. Dadurch läuft das Bieten „symmetrisch“ ab.

Fazit: Es gibt Parameterkombinationen, deren Ergebnisse nicht vorhersehbar sind. Im obigen Set führt nahezu jede Änderung eines der beteiligten Parameter zu völlig neuen Ergebnissen.

4.5.3. Realer und abgerechneter Energieverbrauch

Der Energieverbrauch in den beiden vorangegangenen Kapiteln wurde rein auf der Basis der vom Betriebssystem abgerechneten Energie festgestellt. Dies ist nur zulässig, wenn dieser mit dem tatsächlichen Energieverbrauch übereinstimmt. Da eine Zuordnung zwischen einzelnen Systemaufrufen und Messwerten schwierig bis unmöglich ist, kann nur der gesamte Energieverbrauch der Festplatte mit der abgerechneten Energie (inklusive des Basisenergieverbrauchs) verglichen werden.

Dazu wurden während der Untersuchungen mit den Testsets A und B in Kapitel 4.5.1 parallel Messungen durchgeführt. Aus den gewonnenen Werten wurde der reine Verbrauch der Festplatte ermittelt (siehe auch Anhang A). Dieser wurde mit der Summe der abgerechneten Energie verglichen (die vom Betriebssystemkonto und den Prozesskonten abgebuchte Energie). Dabei fällt auf, das der real gemessene Verbrauch 5% bis 8% niedriger ist, als der abgerechnete. Dieser Wert resultiert hauptsächlich aus Rundungsfehlern, sowie Schwankungen im Energieverbrauch anderer Komponenten des Notebooks, die durch die Messanordnung nicht erkannt werden können. Rundungsfehler treten auf, da im Betriebssystem Energie nicht genauer als 0,1 Joule abgerechnet werden kann.

5. Weiterführende Arbeiten

Da der Zeitraum einer Studienarbeit begrenzt ist, konnte nicht in jeglicher Hinsicht eine optimale Lösung geschaffen werden. In einigen Bereichen könnten weiterführende Arbeiten das Gesamtergebnis verbessern.

Bereits in Kapitel 4.2.4 wurde erwähnt, dass im Filesystem einmal begonnene Aufrufe nur schwer abzubrechen sind. Das Implementieren eines Transaktionskonzepts könnte entscheidende Verbesserungen für die Genauigkeit der Abrechnungen bedeuten. Damit wäre es möglich, nur Systemaufrufe, die auch tatsächlich Plattenzugriffe tätigen, mit Kosten zu belegen. So kann verhindert werden, dass Prozesse zu viel Energie zahlen, daraufhin der Preis sinkt und selbst Aufrufe mit geringem Gebot, auf die Platte zugreifen können ohne warten zu müssen.

In der Praxis hat sich eine weitere Vorgehensweise als ungünstig erwiesen. Während das Konzept ungünstige Spinups verhindert, indem es durch Verzögerungen korrigierend eingreift, sobald das aus energetischer Sicht geboten scheint, wird in der Frage nach ungünstigen Powerdowns auf ein Standardkonzept zurückgegriffen. Der feste Standby Timeout ist hier keine optimale Lösung. [LM99] und [LCS00] stellen eine Vielzahl von Algorithmen, die den festen Timeout ersetzen könnten.

Aber auch durch bessere Nutzung der Komponenten des Konzepts der kooperativen Nutzung kann eine Lösung gefunden werden. Da alle Zugriffe den Marktplatz passieren müssen und dort unter Umständen verzögert werden, könnte der Modusverwalter den Marktplatz für einen „Blick in die Zukunft“ nutzen. Zwar muss aufgrund des Timeouts nicht jedes Gebot zwangsläufig einen Festplattenzugriff bedeuten. Auch ist ein „leerer“ Markplatz kein absolut sicheres Indiz für eine lange Ruhephase. Trotzdem lassen sich zwei Aussagen treffen:

1. Eine stetig steigende Energiesumme auf dem Markplatz kündigt sehr wahrscheinlich Aktionen in näherer Zukunft an.
2. Ein Marktplatzes, auf dem keine Gebote vorliegen, deutet zumindest auf wenig Aktivität hin. In Testsets, die nur aus Autosave−Instanzen bestehen, ist eine lange Ruhephase garantiert. Im hochgradig interaktiven Betrieb treffen Prognosen ohnehin selten zu.

Durch diese Erweiterung könnte der Einspareffekt sicherlich weiter erhöht werden.

6. Zusammenfassung

Ziel dieser Arbeit war es, ein kooperatives Energiesparkonzept für den Zugriff auf die Festplatte zu entwerfen und zu implementieren, dass auf dem Prinzip von Angebot und Nachfrage basiert. Den Prozessen sollte dabei die Möglichkeit gegeben werden, ihren Energieverbrauch beschränken zu können, und damit indirekt die Dringlichkeit der Aufrufe festzulegen.

Dazu wurden in Kapitel 2 umfangreiche Messungen an einer Notebook−Festplatte der Firma IBM vorgenommen. Neben der Energieaufnahme in den einzelnen Modi wurde auch die aufgenommene Leistung während der Übergänge qualitativ erklärt und quantitativ ermittelt. Es stellte sich heraus, dass die bei einem Moduswechsel anfallenden Zusatzkosten so hoch sind, dass mindestens 30,5 Sekunden im Standby Mode verbracht werden müssen. Andernfalls ist es energetisch günstiger, die Platte durchgehend im Low Performance Idle Mode zu betreiben. Darüber hinaus konnte belegt werden, dass das zeitliche Synchronisieren von Festplattenzugriffen ein nicht unbedeutendes Einsparpotential birgt. Bis zu 10 direkt aufeinander folgende Zugriffe sind energetisch kaum aufwendiger als ein einziger.

In Kapitel 3 wurde daraufhin das Konzept der kooperativen Nutzung theoretisch erläutert. Das Layout des Marktplatzes wurde beleuchtet und dessen Funktion als Agent für eine weitere Komponente, den Modusverwalter. Der Modusverwalter nimmt eine zentrale Position ein, und ist die Instanz, die den Energieverbrauch beurteilt (Preisfindung), wie auch regelt (Moduswechsel). Der Energieverbrauch spaltet sich dabei in einen fixen Anteil (Basisenergieverbrauch) und einen aktionsbezogenen Anteil auf. Der Basisenergieverbrauch wird vom Betriebssystem übernommen, während der aktionsbezogene Anteil von den Benutzerprozessen durch Gebote auf dem Marktplatz aufgebracht werden muss. Schließlich galt es noch, die Preispolitik festzulegen. Dabei konnte auf die Ergebnisse des vorherigen Kapitels aufgebaut werden.

Kapitel 4 schließlich befasste sich eingehend mit der Umsetzung und Einbettung in das Betriebssystem MINIX. Aus Sicht der Benutzerprozesse wurde das erweiterte Interface vorgestellt, neue Systemaufrufe, wie auch Veränderungen an schon vorhandenen. Die vorhandene Implementierung des Filesystems stellte sich dabei als problematisch heraus, da ein einmal begonnener Systemaufruf nur schwer wieder abzubrechen ist. Dieses fehlende Transaktionskonzept zwang zu einem Kompromiss, der dazu führt, dass ein Teil der Systemaufrufe unter Umständen unnötig mit Kosten belegt wird. An einem Beispiel wurde das Zusammenspiel der Komponenten erklärt.

Unter Rückgriff auf die Ergebnisse aus Kapitel 2 konnten nun realistische Werte für die energetischen Parameter ermittelt werden. Darauf basierend konnten die Preise, die der Modusverwalters auf dem Marktplatz fordert, genau an die reale Charakteristik der eingesetzten Festplatte angepasst werden. Schließlich wurden zwei Testanwendungen entwickelt, die einerseits zeitkritische Zugriffe, andererseits Zugriffe ohne feste zeitliche Zwänge simulieren. Unterschiedlich parametrierte Instanzen wurden zu Testsets zusammengestellt. Die damit angestellten Untersuchungen lieferten interessante Ergebnisse. So kann abgelesen werden, dass mit dem Konzept der kooperativen Nutzung eine Energieeinsparung gegenüber dem konventionellen Ansatz mit einer reinen Timeout−Regelung erzielt wird. Bezieht man den Basisenergieverbrauch in die ermittelten Werte mit ein, so kann von einer Senkung des Energieverbrauchs der Festplatte um 10% bis 15% gerechnet werden. Im alltäglichen Betrieb können die Werte, je nach Anwendung höher oder niedriger liegen. Weitere Tests mit unterschiedlichen Anteilen an dringenden Zugriffen (Urgentsave) zeigen, dass die Stärken des Konzepts im Bereich der Server− und Hintergrundprozesse liegen. Ein hoher Anteil von dringenden Zugriffen, wie er im interaktiven Betrieb vorkommt, läßt das Konzept zum konventionellen degenerieren. Dagegen sind bei einer

genügend großen Anzahl von Prozessen ohne dringende Zugriffe, große Einsparungen zu erwarten. Lockert man allerdings die strengen zeitlichen Anforderungen im interaktiven Betrieb und toleriert Wartezeiten im Bereich einiger Sekunden, so könnten wohl auch hier größere Einsparungen erzielt werden.

A. Durchführung der Messungen

Für die konkrete Implementierung wurde MINIX auf einem Testsystem installiert. Es handelt sich um ein Notebook IBM Thinkpad 4b 760 ED in dem eine Festplatte vom Typ IBM DTNA−22160 eingebaut ist. Diese ist, wie bei Notebooks üblich, als handliches, einfach austauschbares Modul gekapselt. Leider verhindert diese Bauweise eine direkte Strom− und Spannungsmessung an der Platte.

Deshalb wird stattdessen der Gesamtverbrauch des Geräts gemessen. Um Verfälschungen durch Ladeströme auszuschließen, ist der Akku dabei ausgebaut. Die Messung von Strom und Spannung erfolgt zwischen dem Netzgerät und dem Notebook selbst. Eine speziell angefertigte Steckverbindung führt die notwendigen Kabel an eine Steckerleiste, woran ein Messgerät angeschlossen werden kann.

Als Messgerät wurde ein Conrad M3860−M Digital−Multimeter verwendet. Dieses Gerät kann über eine serielle Verbindung an einen Computer angeschlossen werden, der die Werte protokolliert und auswertet. Für diesen Zweck wird ein Desktop−PC verwendet, der unter Linux 2.2.16 läuft.

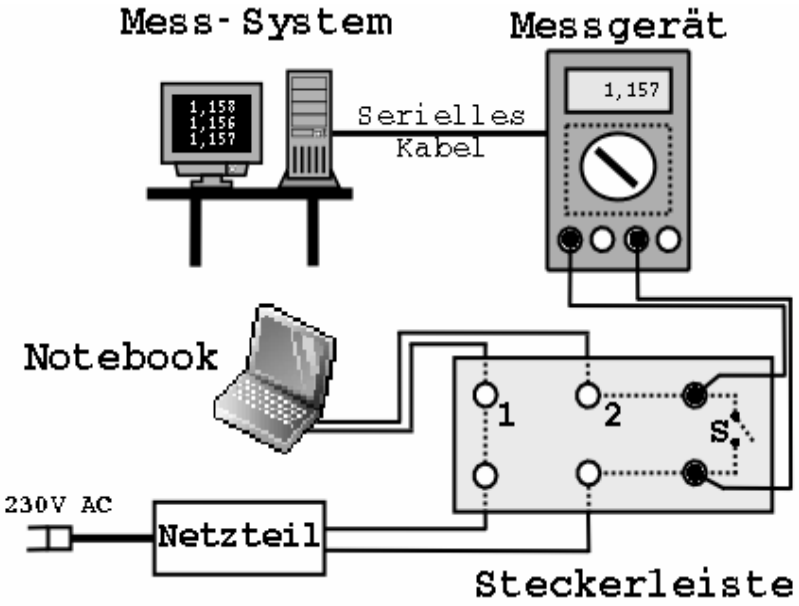

Abbildung 27: Anordnung für die Messung von Strömen

Die in Abbildung 27 gezeigte Anordnung misst die Stromstärke im Sekundärstromkreis nach dem Netzteil. Für die Bestimmung der aufgenommenen Leistung fehlt noch die Spannung, die über dem Notebook abfällt. Hierzu wird der Schalter S geschlossen, das Messgerät auf Spannungsmessung umgestellt und an die Kontakte 1 und 2 angeschlossen. Da leider nur ein Messgerät zur Verfügung steht, müssen Strom− und Spannungsmessungen nacheinander erfolgen, was zu Verfälschungen führen kann, da das Messgerät selbst den Stromkreis geringfügig beeinflusst. Im vorliegenden Strom− und Spannungsbereich ist der Fehler laut Spezifikation des Messgeräts jedoch kleiner als dessen Messgenauigkeit und somit irrelevant.

Um die gemessenen Werte auswerten zu können, werden sie über eine serielle Verbindung auf das Mess−System übertragen und dort protokolliert. Die verwendete Messrate liegt bei vier Messungen pro Sekunde. Sind Ruheleistungen zu messen (keine Änderung von Energiemodi), so wird das Mittel über 250 Messwerte (entspricht 62,5 Sekunden) herangezogen. Um Leistungskurven beim Wechsel der Energiemodi aufzuzeichnen, wird das Mittel aus 10 Einzelkurven errechnet. Angesichts der geringen Abweichungen der Messwertkurven erscheint dies gerechtfertig.

Da aus dem oben genanntem Grund bei allen Messungen stets der Gesamtverbrauch des Note−books statt der Leistungsaufnahme der Platte gemessen wird, muss der Verbrauch der anderen Komponenten bestimmt werden. Dazu wird die Festplatte ausgebaut und eine mittlerer Ruhewert gemessen, wenn MINIX in der gleichen Konfiguration von Diskette gestartet wurde. Die auf dieser Basis errechnete Leistungsaufnahme der Festplatte deckt sich in den inaktiven Betriebs−modi mit der Herstellerspezifikation, was dieses Vorgehen rechtfertigt. In den aktiven Be−triebsmodi verfälscht der zusätzliche Verbrauch des Plattenkontrollers das Ergebnis.

Desweiteren werden Systemprogramme, die zyklisch Festplattenzugriffe tätigen während der Messungen deaktiviert, um keine Verfälschungen zu bekommen.

Literaturverzeichnis

[BKL99] J. Berber, H. Kacher, R. Langer : *Physik in Formeln und Tabellen*, 8. Auflage, Teubner, 1999

[IBM97] IBM: OEM Hard Disk Drive Specification for DTNA−21800/22160, Revision 1.2, 1997, http://www.ibm.com/harddrive

[IBM99] IBM: Adaptive Power Management for Mobile Hard Drives, 1999, http://www.almaden.ibm.com/almaden/mobile_hard_drives.html

[KNU98] Donald E. Knuth: *The Art of Computer Programming Vol. 2*, Addison−Wesley, 1998

[LBM00] Yung−Hsiang Lu, Luca Benini, Giovanni De Micheli: Low−power task scheduling for multiple devices. In *Proceedings of the 8th International Workshop on Hardware/Software Codedesign CODES'2000*, 2000

[LCS00] Yung−Hsiang Lu, Etui−Young Chung, Tajana Simunic, Luca Benini, Giovanni De Michaeli: Quantitative Comparison of power managment algorithms. In *Proceedings of the Conference on Design Automation and Test in Europe DATE'2000*, 2000

[LM99] Yung−Hsiang Lu, Giovanni De Michaeli: Adaptive Hard Disk Power Managment on personal computers. In *Proceedings of the IEEE Great Lakes Symposium*, pages 50−53, 1999

[Tan97] Andrew S. Tanenbaum, Albert S. Woodhull: *Operating Systems: Design and Implementation*, Second Edition, Prentice−Hall, 1997