Real-Time Operating Systems – Ein Überblick

Stefan Tittel
Universität Dortmund

Proseminar: Werkzeuge und Techniken zur Spezifikation,
Simulation und Implementierung von eingebetteten Systemen, 2004

Inhaltsverzeichnis

1 Einführung

1.1 Operating Systems

Um im Weiteren über Operating Systems (und Real-Time Operating Systems im Besonderen) sprechen zu können, wird der Begriff des Operating System zunächst definiert.

Definition 1.1 (Operating System)
Das Operating System *ist das Programm, welches beim Start eines Rechners geladen wird. Es dient als eine Schnittstelle zwischen den Rechnerkomponenten wie Hardware und BIOS und den Anwendungen, die auf dem Rechner laufen. Es bietet außerdem grundlegende Funktionen für die Verwaltung und Pflege des Operating System und Dateisystems.*

Das Operating System ist folglich die zentrale Instanz, die nach der ersten Initialisierung durch das BIOS die vollständige Kontrolle über den Rechner übernimmt. Jede auf einem System laufende Anwendung benötigt unterschiedliche Ressourcen, seien es Rechenzeit, Arbeitsspeicher, Festspeicher oder Peripheriegeräte. Die Verteilung dieser Ressourcen nimmt das Operating System vor; ferner abstrahiert es in der Regel Hardware in Form von Schnittstellen. Eine Anwendung, die eine bestimmte Hardware nutzen möchte, muss so nur mit der vom Operating System bereitgestellten Schnittstelle kommunizieren können und nicht die Sprache der Hardware selbst sprechen. Überdies bieten viele Operating Systems grundlegende Sicherheitsfunktionen an, wie z. B. die Verwaltung von Lese-, Schreib- und Ausführungsrechten von Dateien und Verzeichnissen und eine Zugriffskontrolle der Hardwarenutzung. Darüber hinaus wird i. d. R. eine Schnittstelle zur Interaktion mit dem Benutzer zur Verfügung gestellt.

Unter üblichen Einsatzbedingungen wird vom Operating System verlangt, bei der Ausführung seiner Aufgaben gleichermaßen effizient und reaktionsschnell zu sein. Eine Hardwareabstraktion nützt bspw. wenig, wenn sie so aufwändig ist, dass kaum noch Leistung für Anwendungen übrig bleibt. Auch ein effizientes Scheduling wird dem Nutzer wenig Freude bereiten, sollte er minutenlang auf die Reaktion des Systems warten müssen.

1.2 Notwendigkeit eines Real-Time Operating System

Im vorherigen Abschnitt wurde festgestellt, dass der Aufgabenbereich eines Operating System im Allgemeinen sehr komplex ist und ein Operating System im Regelfall auf ein möglichst gutes Durchschnittsverhalten hin optimiert ist. Das Verhalten des Operating Systems in Bezug auf die Verteilung von Systemressourcen ist daher praktisch nicht temporal determinierbar. Ein in gewissen Grenzen festgelegtes zeitliches Verhalten ist jedoch manchmal wünschenswert, wie folgende Beispiele aus dem Bereich der eingebettenen System verdeutlichen sollen:

- *Der Ticketautomat an der Bahnhaltestelle.* Ein Benutzer des Ticketautomaten dürfte mindestens verlangen, dass er sein Ticket vor Abfahrt seines Zuges erhält.

- *Das Anti-Blockier-System im Kraftfahrzeug.* Durch kurzes periodisches Lösen der Bremse die Lenkfähigkeit des Fahrzeugs aufrecht zu erhalten, ergibt nur *vor* dem möglichen Unfall Sinn.
- *Die Notabschaltung im Kernkraftwerk.* Die Auslösung der Kernabschaltung nach bereits erfolgter Kernschmelze ist wenig hilfreich.
- *Der Drucker im Büro.* Bis zum spätest möglichen Zeitpunkt des Dokumentenversandes muss das Dokument gedruckt worden sein.

Die Korrektheit eines herkömmlichen Systems hängt von der Korrektheit der ausgeführten Berechnung ab. Würde dieser Maßstab auch für obige Beispiele gelten, so verhielte sich die Reaktorabschaltung auch dann korrekt, wenn sie erst zwei Wochen nach der Kernschmelze die Abschaltung initiierte. Offenbar ist dieser Korrektheitsbegriff nicht in allen Fällen ausreichend. Es wird daher der Begriff des *Real-Time System* eingeführt.

Definition 1.2 (Real-Time System)
Ein Real-Time System *ist ein System, dessen Korrektheit nicht nur korrekte Berechnungen verlangt, sondern darüber hinaus zeitliche Vorgaben über das Eintreffen des Ergebnisses macht. Ein Real-Time System reagiert auf zeitlich vorhersehbare Weise auf das Eintreten externer Stimuli.*

Ein *Real-Time Operating System* sei dann wie folgt definiert:

Definition 1.3 (Real-Time Operating System)
Ein Real-Time Operating System *ist ein Operating System mit den notwendigen Eigenschaften zum Betrieb eines Real-Time System.*

1.3 Klassifizierung von Real-Time Systems

Wie bereits die Beispiele aus Abschnitt 1.2 vermuten ließen, variiert die mögliche Schadwirkung eines temporalen Fehlverhaltens von Real-Time System zu Real-Time System. Es mag für den Ticketkäufer zwar ärgerlich sein, wenn er länger als gewünscht am Ticketautomaten verweilen muss, eine nahegelegende Reaktorschmelze dürfte jedoch weitaus unangenehmere Konsequenzen nach sich ziehen.

Die Konsequenzen im Versagensfall bilden die Grundlage für die folgende Klassifizierung von Real-Time Systems.

Definition 1.4 (Soft Real-Time System)
Stellt in einem System das gelegentliche Verpassen von Deadlines eine akzeptable Qualitätsminderung dar, so handelt es sich um ein Soft Real-Time System.

Aufgrund dieser relativ moderaten Anforderung an das zeitlich deterministische Verhalten des Operating System, genügt es oft, bestehende Operating Systems leicht zu modifizieren und so ein Real-Time Operating System für Soft Real-Time Systems zu erhalten. Der Ticketautomat und der Drucker aus Abschnitt 1.2 sind Beispiele für Soft Real-Time Systems.

Definition 1.5 (Firm Real-Time System)
Stellt in einem System das Verpassen von Deadlines eine nicht akzeptable Qualitätsminderung dar, so handelt es sich um ein Firm Real-Time System.

Bei einem Videorekorder handelt es sich beispielsweise um ein solches Firm Real-Time System. Ein Videorekorder, der nicht in der Lage ist eine Aufnahme zuverlässig rechtzeitig zu starten und zu stoppen, ist wertlos, die Größenordnung einer Reaktorschmelze wird aber auch hier nicht annähernd erreicht.

Definition 1.6 (Hard Real-Time System)
Wenn das Verpassen einer Deadline katastrophale Folgen nach sich zieht, handelt es sich um ein Hard Real-Time System.

Aufgrund dieser sehr strikten Anforderung ist die Modifikation eines herkömmlichen Operating System zu einem Real-Time Operating System für Hard Real-Time Systems deutlich schwieriger. Beispiele für Hard Real-Time Systems sind das Anti-Blockier-System und die Reaktornotabschaltung aus Abschnitt 1.2.

2 Scheduling

2.1 Grundlagen und Definitionen

Wie im vorherigen Abschnitt erläutert wurde, besteht die Besonderheit eines Real-Time Operating System darin, Systeme zu betreiben, die (in gewissen Grenzen) temporal determinierbar reagieren müssen. Der unter den Aufgaben des Operating System davon besonders betroffene Bereich ist das Scheduling, mit dem sich dieser Abschnitt beschäftigen wird. Bevor in Abschnitt 2.2 einzelne Scheduling-Algorithmen näher vorgestellt werden, sollen zunächst die dafür notwendigen grundlegenden Begriffe definiert und erläutert werden.

Definition 2.1 (Worst-Case Execution Time)
Die Worst-Case Execution Time (WCET) *ist die obere Grenze der Ausführungszeit eines Tasks.*

Zu wissen, wie lange ein Task im ungünstigsten Falle zur Ausführung benötigt, ist aus naheliegenden Gründen sehr günstig. Ein Scheduling-Algorithmus wird im Regelfall die genaue Ausführungzeit eines Tasks nicht im Voraus kennen. Um zu wissen, wann nun welcher Task gestartet werden muss, um alle Deadlines einzuhalten, ist eine Kenntnis der Ausführungszeit jedoch mehr als nützlich. Statt der tatsächlichen (meist unbekannten) Ausführungszeit, wird daher oft die Worst-Case Execution Time zur Berechnung herangezogen.

Aber auch deren Berechnung gestaltet sich nicht immer einfach. Moderne Rechnerarchitekturen mit Caches und Pipelines verhalten sich bereits selbst nicht mehr praktisch temporal determinierbar. Manche Tasks verfügen darüber hinaus nicht über eine endliche Worst-Case Execution Time, z. B. kann die Bedingung einer bedingten Schleife unendlich lange erfüllt sein, so dass diese nie terminiert.

Definition 2.2 (Periodizität von Tasks)
Ein Task, der alle p Zeiteinheiten ausgeführt wird, heißt periodisch*; p ist dabei seine* Periode*. Tasks, die nicht periodisch sind, heißen* nicht-periodisch*.*

Nicht-periodische Tasks, die zu nicht vorhersehbaren Zeitpunkten eintreffen, heißen sporadisch*.*

Ein Beispiel für einen periodischen Task stellt die Temperaturüberprüfung der Reaktorabschaltung dar, welche in regelmäßigen Intervallen die Temperatur am Sensor ausliest und mit vorgebenen Sollwerten vergleicht. Sporadisch hingegen ist z. B. die Benutzerschnittstelle des Ticketautomaten.

Definition 2.3 (Dynamisches und statisches Scheduling)
Werden die Scheduling-Entscheidungen während der Laufzeit getroffen, so handelt es sich um dynamisches Scheduling*.*

Werden die Scheduling-Entscheidungen bereits zur Zeit der Entwicklung festgelegt, handelt es sich um statisches Scheduling*.*

Systeme mit Zeitgeber, die Tasks ausschließlich nach Start- und Stoppzeiten schedulen, heißen entirely time triggered (TT)*.*

Ein dynamischer Scheduler kennt in der Regel den Ressourcenbedarf und die Abhängigkeiten von Tasks untereinander nicht, im Gegensatz zum statischen Scheduling, bei welchem der Entwickler diese Dinge bei der Aufstellung eines Schedules berücksichtigen kann.

TT-Systeme stellen eine Untermenge der Systeme mit statischem Scheduling dar.

Definition 2.4 (Kooperatives und präemptives Scheduling)
Nimmt der Scheduler einen Kontextwechsel nur dann vor, wenn der laufende Task dies explizit anbietet oder beendet ist, handelt es sich um kooperatives Scheduling.

Nimmt der Scheduler einen Kontextwechsel vor, ohne dass die Bedingungen für kooperatives Scheduling erfüllt sind, handelt es sich um präemptives Scheduling.

Der Vorteil des kooperativen Schedulings liegt vor allem in der vergleichsweise einfachen Implementierung. Seine Nachteile liegen jedoch in einem sehr schlechten Reaktionsverhalten, sowohl auf externe Stimuli (z. B. Benutzereingaben), als auch auf die Neuankunft von Tasks. Ist die Neuankunft eines Tasks während der Ausführung eines anderen erlaubt, können Deadlines nicht mehr sicher eingehalten werden. In solchen Fällen ist daher in jedem Fall präemptives Scheduling von Nöten.

Beispiele für kooperatives Scheduling:

- Windows $\leq$ 3.11
- Videorekorder

Beispiele für präemptives Scheduling:

- Windows $\geq$ 95
- Linux
- OS/2
- Solaris
- BetaNova

Es findet ferner eine Unterscheidung zwischen Online- und Offline-Scheduling statt. Beim Offline-Scheduling kennt der Scheduling-Algorithmus die Tasks, sowie ggf. deren Deadline und Execution Time bereits im Voraus, und kann dieses Wissen zur Planung des Schedules nutzen. Ein Online-Scheduler hingegen sieht Tasks und deren Eigenschaften erst ab dem Zeitpunkt, zu dem sie eintreffen.

Darüber hinaus wird zwischen Ein- und Mehrprozessor-Scheduling und Scheduling mit abhängigen und unabhängigen Tasks unterschieden.

Weiterhin sei definiert:

Definition 2.5 (Laxity)
Die Differenz zwischen der Ausführungszeit eines Tasks und dem Deadline-Intervall heißt Laxity*.*

Definition 2.6 (Maximum Lateness)
Die Differenz zwischen der Fertigstellungszeit und der Deadline – maximiert über alle Tasks – heißt Maximum Lateness*. Wenn kein Task seine Deadline überschreitet, ist die Maximum Lateness negativ.*

Definition 2.7 (Schedulability)
Eine Menge von Tasks heißt schedulable*, wenn ein Schedule der Form existiert, dass jeder Task seine Deadline erfüllt.*

2.2 Algorithmen

Alle im Folgenden besprochenen Algorithmen sind optimal hinsichtlich der Schedulabilty, einige darüber hinaus optimal in Hinblick auf eine Minimierung der Maximum Lateness. Ferner handelt es sich nur um eine kleine Auswahl der für Scheduling in Real-Time Systems verfügbaren Algorithmen.

2.2.1 Earliest Due Date (EDD)

Der EDD-Algorithmus setzt voraus, dass Tasks nicht-periodisch und voneinander unabhängig sind, sowie gleichzeitig eintreffen. Er ist ein Scheduling-Algorithmus für Einprozessorsysteme und implementiert kooperatives Scheduling.

Vorgehensweise:

1. Sortiere alle Tasks nach Deadline aufsteigend.
2. Führe die Tasks gemäß dieser Sortierreihenfolge aus.

Die Korrektheit des Algorithmus ist sehr anschaulich zu erfassen: Würde nicht der Task mit der am nächsten liegenden Deadline ausgeführt, so könnte dieser Task seine Deadline verletzen, obwohl durch seine rechtzeitige Ausführung eine Einhaltung der Deadline möglich gewesen wäre (die Taskmenge also schedulable ist). Ist die Deadline des Tasks mit kürzester Deadline hingegen auch bei frühstmöglicher Ausführung nicht einzuhalten, oder verfehlt der Task mit zweitkürzester Deadline seine Deadline, weil der Task mit kürzester Deadline zuerst ausgeführt wurde, dann ist die Menge von Tasks ohnehin nicht schedulable und der Algorithmus trägt nicht die Versagensschuld. Ein ausschließlich mit Sortierkriterium arbeitender Algorithmus muss also unter diesen Vorbedingungen immer den Task mit der am nächsten liegenden Deadline auswählen.

Sind die Deadlines aller Tasks im Voraus bekannt, so kann der Earliest Due Date-Algorithmus statisch implementiert werden. Der EDD-Algorithmus ist einfach zu implementieren und darüber hinaus sogar optimal in Hinblick auf eine Minimierung der Maximum Lateness. Leider schränken die strengen Vorbedingungen seinen Einsatzbereich sehr ein.

2.2.2 Earliest Dateline First (EDF)

Die Vorbedingungen des EDF-Algorithmus bestehen wie auch beim EDD-Algorithmus in nicht-periodischen, voneinander unabhängigen Tasks. Auch beim EDF-Algorithmus handelt es sich um einen Algorithmus für Einprozessorsysteme. Im Unterschied zum EDD-Algorithmus können Tasks jedoch zu beliebiger Zeit eintreffen, weshalb der EDF-Algorithmus präemptives Scheduling implementiert.

Vorgehenweise:

1. Führe immer den Task mit der am nächsten liegenden Deadline aus (ähnlich *EDD*).
2. Bei Neuankunft eines Tasks mit früherer Deadline, wechsele zu diesem.

Wenn ein neuer Task eintrifft, so wird er in eine nach Deadline aufsteigend sortierte Warteschlange zur Ausführung bereiter Tasks eingereiht. Wird der neue Task an die erste Position dieser Warteschlange eingefügt, so wird der laufende Task unterbrochen und der neue Task ausgeführt.

Der EDF-Algorithmus kann als Erweiterung des EDD-Algorithmus angesehen werden. Er ist dynamisch und ebenfalls optimal in Hinblick auf die Minimierung der Maximum Lateness. Da Tasks im Gegensatz zum EDD-Algorithmus nun nicht mehr alle gleichzeitig eintreffen müssen, gibt es für den EDF-Algorithmus deutlich mehr Einsatzbereiche.

2.2.3 Least Laxity (LL)

Der LL-Algorithmus setzt ebenfalls nicht-periodische, voneinander unabhängige Tasks voraus. Zusätzlich verlangt er Kenntnis über die Ausführungszeiten (bzw. die WCETs) aller Tasks. Es handelt sich um einen präemptiven, dynamischen Algorithmus für ein Einprozessorsystem, welcher wie folgt vorgeht:

1. Führe immer den Task mit der geringsten Laxity aus.

Um dies zu erreichen, werden Task-Prioritäten als monoton fallende Funktion der Laxity genutzt. Die dadurch entstehenden dynamischen Prioritäten sind in vielen Operating Systems nicht implementiert. Die Korrektheit ist wie bei den beiden vorhergehenden Algorithmen ebenfalls sehr leicht zu veranschaulichen: Es wird immer der Task ausgeführt, der am ehesten Gefahr läuft seine Deadline zu verletzen, dessen Spielraum/Laxity also am geringsten ist. Wird trotzdem die Deadline verletzt, so war die Menge der Tasks nicht schedulable und der Algorithmus trägt keine Schuld.

Der LL-Algorithmus kann u. U. recht viele Kontextwechsel erzwingen. Vor allem sind es aber das zusätzliche Wissen um die Ausführungzeiten der Tasks und die aufwändigere Implementierung, die den LL-Algorithmus nur selten eine Alternative zum ELF-Algorithmus werden lassen.

2.2.4 Latest Deadline First (LDF)

Im Gegensatz zu den drei vorherigen Algorithmen behandelt der LDF-Algorithmus das Scheduling von untereinander abhängigen Tasks. Die Tasks müssen nicht-periodisch sein, gleichzeitig eintreffen und die Länge ihrer Ausführungszeit muss im Voraus bekannt sein. Der LDF-Algorithmus implementiert kooperatives, dynamisches Scheduling für Einprozessorsysteme.

Vorgehensweise:

1. Liegen die Abhängigkeiten als Halbordnung vor, erzeuge mittels topologischer Sortierung eine vollständige Ordnung.
2. Führe die Tasks in der Reihenfolge der vollständigen Ordnung aus.

Bei der topologischen Sortierung werden im Falle einer vorliegenden Halbordnung zuerst alle Tasks ohne Abhängigkeiten in eine Liste überführt. Danach geschieht gleiches mit allen Tasks, deren Abhängigkeiten ausschließlich aus bereits in dieser Liste vorhandenen Tasks bestehen.

Der LDF-Algorithmus ist optimal in Hinblick auf die Minimierung der Maxium Lateness.

2.2.5 Rate Monotonic Scheduling (RM)

Der RM-Algorithmus stellt Scheduling für periodische Tasks bereit. Die für ihn zu erfüllenden Voraussetzungen sind zahlreich:

- Periodische, voneinander unabhängige Tasks
- Deadline und Periodendauer bei jedem einzelnen Task gleich lang
- Länge alle Ausführungszeiten im Voraus bekannt
- Es gilt $\sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{\frac{1}{n}} - 1)$, wobei $i \in \{\text{Tasks}\}$, c_i Ausführungszeit, p_i Periodendauer und n Anzahl der Tasks.

Der linke Teil der Formel stellt die aufsummierte Auslastung dar, welche ohnehin nie größer werden darf als die Anzahl der im System verfügbaren Prozessoren. Dies liegt sehr nahe, denn ein Prozessor kann maximal einen fortwährend laufenden Task A ($c_A = p_A$) ausführen, oder maximal zwei Tasks A, B mit $c_A + c_B = p_A + p_B$ oder maximal drei Tasks A, B, C mit $c_A + c_B + c_C = p_A + p_B + p_C$ usw. Systeme mit mehreren Prozessoren leisten das dementsprechend Mehrfache.

Die obige Voraussetzung schränkt jedoch weiter ein: Die aufsummierte Auslastung muss kleiner oder gleich $n(2^{\frac{1}{n}} - 1)$ sein, d. h. eine volle Auslastung des Systems durch Tasks ist bei Verwendung dieses Algorithmus nicht möglich.

Der RM-Algorithmus geht wie folgt vor:

1. Vergib Prioritäten nach Periodenlänge: Je kürzer p_i desto höher die Priorität.

Der RM-Algorithmus implementiert präemptives Scheduling mit statischen Prioritäten. Letzteres vereinfacht seine Implementierung in herkömmliche Operating Systems enorm.

3 Embedded Linux

3.1 Motivation

Linux ist ein quelloffenes, freies Operating System, welches für viele verschiedene Rechnerarchitekturen[1] verfügbar ist. Für Linux existieren ferner Compiler für fast alle Programmiersprachen, sowie eine Vielzahl an verwendbaren Bibliotheken und Programmen. Ebenso wird Linux bereits in einer Vielzahl eingebetteter System eingesetzt und hat sich als zuverlässig erwiesen. Anstatt ein neues Real-Time Operating System und die zum sinnvollen Betrieb notwendigen Compiler, Bibliotheken und Anwendungen von Grund auf neu zu entwickeln, kann durch einen Rückgriff auf bereits Vorhandenes sehr viel Aufwand gespart werden.

3.2 Bestandsaufnahme

3.2.1 Linux als Betriebssystem für Soft Real-Time Systems

- Linux unterstützt die Vergabe von statischen Prioritäten.
- Es sind Kernelpatches mit alternativen Schedulern verfügbar.
- Dank offener Quellen ist die Entwicklung eines Schedulers mit dynamischen Prioritäten denkbar.

3.2.2 Linux als Betriebssystem für Hard Real-Time Systems

- Mangelnde Scheduling-Fähigkeiten
- Ungeeignetes Interrupt-Handling
- Optimierung auf Effizienz, nicht auf Einhaltung von Deadlines

3.3 Vom Non-Real-Time zum Real-Time Operating System

Wie in 3.2.2 gesehen, taugt Linux als Operating System für Hard Real-Time Systems nicht. Die Grundidee trotzdem die Vorteile von Linux mit den Ansprüchen an ein Hard Real-Time System zu verknüpfen, besteht darin, das System von einem Real-Time Kernel kontrollieren zu lassen, in dessen Rahmen der Linux-Kernel als Idle-Task läuft. Die Hardware- und Interruptverwaltung wird dabei vollständig vom Real-Time Kernel übernommen. Neben dem üblichen Kernel- und User-Space, erhält das System einen zusätzlichen Real-Time-Space, in welchem sämtliche Task laufen, die ein Hard Real-Time Verhalten verlangen. Sämtliche Hardwareanfragen und Interrupt-Anforderungen des Linux-Kernels werden

[1] Compaq Alpha AXP, Sun SPARC und UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, DEC VAX, AMD x86-64 und CRIS. Quelle: [kernel.org].

dabei vom Real-Time Kernel abgefangen, um eine negative Beeinflussung des Real Time-Verhaltens durch den Linux-Kernel auszuschließen.

Beispiele für eine solche Implementierung von Linux als Hard Real-Time System stellen RT-Linux und RTAI dar.

Die Vorteile des Linux-Einsatzes lassen sich bei Tasks, die im Real-Time Space laufen, nicht nutzen. Das heißt insbesondere, dass Speicherschutzfunktionen, Debugger und Interprozess-Kommunikation (z. B. durch FIFOs, Shared Memory, Mailboxes oder Messages) selbst realisiert werden müssen.

Literatur

[1] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers, Boston, 2003

[2] Raquel S. Whittlesey-Harris [online]. *Real-Time Operating Systems*. Internet: `<http://tinyurl.com/3dgb6>`, 2001 [Stand 2004-04-10]

[3] Robert Schwebel. *Embedded Linux – Handbuch für Entwickler*. mitp-Verlag, Bonn, 2001

[4] Rüdiger Brause. *Betriebssysteme – Grundlagen und Konzepte*, 2. Aufl., Springer-Verlag, Berlin/Heidelberg, 2001

[5] Mathai Joseph, Alan Burns, Andy Wellings, Krithi Ramamritham, Jozef Hooman, Steve Schneider, Zhiming Lui, Henk Schepers [online]. *Real-time Systems Specification, Verification and Analysis*. Internet: `<http://www.tcs.com/techbytes/htdocs/RTSbook.zip>`, 2001 [Stand 2004-04-10].

[kernel.org] Kernel.Org Organization, Inc [online]. *The Linux Kernel Archives* `<http://www.kernel.org>` 2004 [Stand 2004-05-14].