Aufbau und Implementierung des HydrixOS Kernels *hymk*

Friedrich Gräter

26. Januar 2007

Lizenz

Vorwort

Das Ziel des HydrixOS-Betriebssystemprojekt soll ein in jeder Hinsicht offenes, flexibles, modulares, stabiles, zuverlässiges, portables und architekturtransparentes Betriebssystem sein, das leistungsfähig und in seiner Arbeitsweise professionell und verständlich organisiert ist.

Um dieses doch sehr hochgesteckten Ziel zu erreichen, müssen bereits auf unterster Ebene die Grundsteine richtig gelegt werden. Daher ist ein leistungsfähiger und möglichst einfacher Mikrokernel unbedingt erforderlich.

Der neue HydrixOS-Kernel, der den Namen *hymk* (hymk - HydrixOS μ-Kernel) trägt, soll ein vollständiger Mikrokernel sein, der auf einer extrem vereinfachten, synchronen und paging-basierten IPC aufbaut. Der neue Kernel soll grundsätzlich an die Standardaufgaben eines Client-Server-Systems orientiert werden, bei dem im wesentlichen die Interaktion zwischen Clients und Servern über virtuelle Dateien geschieht. Portabilität ist bei diesem Kernel mehr Sache der vom Kernel angebotenen Konzepte, als eine Sache der Schnittstellen an sich. Gleichzeitig soll er aber auch alles bieten, um Architekturtransparenz zu realisieren - nur bietet er jetzt nur noch Mechanismen zur Unterstützung von dieser an. Die eigentliche Umsetzung der Virtualisierungssoftware, CPU-Emulatoren und Programmübersetzern soll nun vollständig im Benutzer-Modus stattfinden. Zu diesen Unterstützungsfeatures gehört unter anderem die Möglichkeit Threads vollständig fernzusteuern und deren Systemaufrufe und Exceptions an andere Threads im Benutzer-Modus weiterzuleiten.

Diese Dokument soll einen Überblick über die grundlegenden Konzepte des Betriebssystem bieten. Es setzt dabei gewisse Grundkenntnisse über die üblichen internen Strukturen von Betriebssystemen voraus. Ferner wird schließlich die auf der x86-Implementierung verwendete Schnittstelle erläutert. Im letzten Teil wird dann die konkrete Implementierung des Kernels dokumentiert. Das Dokument sollte in der vorgesehenen Reihenfolge gelesen werden, da jeder Teil die jeweils vorausgehenden Teile des Dokuments voraussetzt.

Inhaltsverzeichnis

Teil I

Architektur des hymk

Kapitel 1

Grundlegende Konzepte

1.1 Virtueller Adressraum

1.1.1 Klärung des Begriffs

Eines der grundlegensten Konzepte des *hymk* ist das Konzept des *virtuellen Adressraums* (*virtual address space*). Ein virtueller Adressraum ist ein kontinuierlicher Adressraum, der aus Speicherseiten (*pages*) fester, plattformabhängiger Größe besteht (auf x86-Plattformen sind es 4 KiB). Er ist deshalb virtuell, weil Speicherung und Zusammensetzung dieses Adressraums mit keinem *physikalischen Adressraum* (z.B. dem RAM) übereinstimmen muss. Der Inhalt jeder Speicherseite kann an einem völlig beliebigen Ort gespeichert sein, der in keiner Beziehung mit den restlichen Speicherseiten des virtuellen Adressraums steht. So können zwei Seiten, die in einem virtuellen Adressraum direkt nebeneinander liegen, an völlig unterschiedlichen, weit auseinander liegenden Plätzen im physikalischen Adressraum gespeichert sein.

Ein solcher Speicherort, der den Inhalt einer Speicherseite enthält, wird als *Seitenrahmen* (*page frame*) bezeichnet. Normalerweise liegen diese Seitenrahmen irgendwo über das RAM des Rechners verteilt.

Sie können aber rein prinzipiell auch auf eine Festplatte ausgelagert werden, um einen *virtuellen Arbeitsspeicher* zu realisieren. Dazu werden die Seiten speziell markiert, so dass ein Zugriff auf diese automatisch an einen Systemdienst - den sog. *Paging Daemon* (*hyPageD*) weitergeleitet wird, damit dieser die Speicherseite wieder in das RAM zurückholen kann.

Prinzipiell ist jeder virtueller Adressraum von einem anderen unabhängig und somit vor unerlaubten Zugriff von Programmen geschützt, die nicht in diesem Adressraum arbeiten. Um jedoch einen schnellen Datenaustausch möglich zu machen, können Seitenrahmen von mehreren virtuellen Adressräumen gemeinsam genutzt werden. Dieses Verfahren wird öfters als „*memory sharing*“ oder „*shared memory*“ bezeichnet.

Zum Schluß muss eine Speicherseite nicht zwingend einem Seitenrahmen zugewiesen werden. Normalerweise hat ein virtueller Adressraum eine Größe von 4 GiB (2^{32} Byte) oder gar von 2^{64} Bytes, so dass auf den derzeit verfügbaren Architekturen gar nicht genug Seitenrahmen zur Verfügung ständen, um den Adressraum auszufüllen. Daher werden die meisten Seiten eines virtuellen Adressraums gar nicht mit einem Seitenrahmen verknüpft sein. Ein Zugriff auf solche unverknüpften Speicherseiten führt meist zu einer Exception.

Da aber natürlich ein normales Programm nichts von Speicherseiten und Seitenrahmen wissen will und einfach nur auf seinen Speicher zugreifen können will, muss das Konzept des virtueller Adressraums bereits auf der Ebene der CPU durch eine sog. *Memory Managment Unit* (*MMU*) unterstützt werden. Die Aufgabe einer MMU ist die transparente Übersetzung virtueller Adressen in physikalische Adressen bei jedem Speicherzugriff eines Programms (um dies zu beschleunigen gibt es sog. *Translation Lookaside Buffers*, die Übersetzungen zwischenspeichern). Der virtuelle Adressraum ist

also ein Konzept, die bereits auf Hardwareebene umgesetzt wird und vom Kernel nur noch gesteuert werden muss.

1.1.2 Verwaltung virtueller Adressräume

Ein virtueller Adressraum wird über sog. *Seitentabellen* (*page tables*) verwaltet. In diesen Seitentabellen sind für jede Speicherseite *Seitendeskriptoren* (*page descriptors*) enthalten, die die Adresse des Seitenrahmens angeben, der mit der jeweiligen Speicherseite assoziiert ist. Außerdem enthalten sie eine Reihe weiterer Informationen, die spezielle Zugriffsrechte auf die Seite definieren.

Da in einem normalen virtuellen Adressraum der größte Teil des Speichers ungenutzt bleibt, wäre es Speicherverschwendung, trotzdem für alle nicht benötigten Speicherseiten eigene Seitendeskriptoren zu speichern. Statt dessen organisiert man die Tabelle mit Hilfe eines zwei- oder mehrstufigen Verfahrens. An der Spitze eines zweistufigen Modells steht ein *Seitenverzeichnis* (*page directory*), das z.B. für alle 4 MiB (1024 Speicherseiten bei 4 KiB Seitengröße) die Adresse einer Seitentabelle beschreibt, die dann die tatsächlichen Deskriptoren für die jeweiligen Speicherseiten ihres Bereichs enthält. Wenn ein größerer Speicherbereich nicht genutzt wird, müsen für ihn weder Speicherseiten, noch Seitentabellen reserviert werden, da der Eintrag für die jeweilige Seitentabelle im Seitenverzeichnis einfach leer gelassen wird.

Um die interne Organisation und Verwaltung eines virtuellen Adressraums kümmert sich alleine der Kernel, da die damit verbundenen Strukturen zu sehr plattformabhängig sind. Für den Programmierer existieren weder Seitentabellen, noch Seitenverzeichnisse. Wenn er mit der virtuellen Speicherverwaltung überhaupt in Kontakt tritt, dann nur dann, wenn es um die Zugriffsflags geht, die für Speicherseiten gesetzt werden können. Diese Flags sind plattformunabhängig definiert - auch wenn sie nicht zwingend auf jeder Plattform umgesetzt werden können. Derzeit sind es folgende Flags[1]:

- Lesezugriff erlaubt
- Schreibzugriff erlaubt (bei einigen Plattformen, wie x86, setzt Schreibzugriff Lesezugriff voraus)
- Ausführung erlaubt (auf x86 nicht seperat definierbar)
- Schreibzugriff nur für den Kernel erlaubt
- Speicherseite ist für das *Copy-On-Write*-Verfahren selektiert
- Speicherseite ist vom Caching durch die CPU ausgeschlossen

1.1.3 Aufbau eines virtuellen Adressraums

Je nach Plattform ist der Aufbau des virtuellen Adressraums vorgeben. Er teilt sich normalerweise in einen Kernel-Adressraum und einen Benutzer-Adressraum. Der Benutzer-Adressraum kann in jedem virtuellen Adressraum völlig unterschiedlich zusammengesetzt sein. Der Kernel-Adressraum bleibt in allen virtuellen Adressräumen von seiner Zusammensetzung her identisch und enthält Kernel-Code und Daten, sowie die sogenannten *Info-Pages*. Die Info-Pages sind ein Bereich des Kernel-Adressraums, der Informationen über das Systems und alle Systemdeskriptoren enthält. Dadurch können Programme Systeminformationen ohne einen Systemaufruf aus dem Benutzer-Modus heraus abrufen.

[1] Es sei darauf hingewiesen, dass derzeit weitere Flags, die für die Implementierung eines Seitenauslagerungsmechanismuses benötigt werden könnten (z.B. Zugriffsprüfung) bewusst noch nicht mit aufgenommen wurden, da die Seitenauslagerung unter HydrixOS erst zu späterer Zeit implementiert werden kann und voreilige Festlegungen vermieden werden sollten.

Ferner ist meist die erste und die letzte Speicherseite jedes Adressraums komplett gesperrt, um Null-Pointerzugriffe leichter aufdecken zu können.

1.2 Subjekte

1.2.1 Begriffsklärung

Ein Subjekt ist unter HydrixOS alles, das als solches Aktionen an Systemobjekten ausführen kann oder in dessen Namen Aktionen ausgeführt werden können. Der *hymk* kennt grundsätzlich vier Sorten von Subjekten:

- Platzhalter (Typ 0)
- Threads (Typ 1)
- Prozesse (Typ 2)
- Prozessgruppen (Typ 4)

Jedes Subjekt wird durch seine einmalige *Subjekt-ID* (*SID*) idendifiziert. Diese Subjekt-ID ist eine 32-Bit-Zahl, die einen 8-bit Header enthält, der den Typ des Subjekts beschreibt (siehe obige Typnummer). Die restlichen 24-Bit der Zahl enthalten die Nummer des jeweiligen Subjekts, wobei die Nummern der unterschiedlichen Typen sich überschneiden dürfen. Die Anzahl der möglichen SIDs ist bei Threads und Prozessen plattformmäßig beschränkt (bei x86 auf 4096).

1.2.2 Platzhalter

Das System definiert als Subjekte drei Platzhalter, die immer dann verwendet werden, wenn eine ganz besondere Art von Subjekt-Gruppe symbolisch dargestellt werden soll:

- Das *invalid subject* (`0x00FFFFFF`) wird immer dann verwendet, wenn eine ungültige Subjekt-ID angegeben werden soll.
- Das *null subject* (`0x00000000`) wird immer dann verwendet, wenn als Subjekt-ID ein gültiges, aber rechteloses Subjekt definiert werden soll.
- Das *kernel subject* (`0x00000001`) wird immer dann verwendet, wenn der Kernel als Subjekt auftritt. Da der Kernel weder als Thread, noch als Prozess auftreten kann, besitzt er einen eigenen Platzhalter.

1.2.3 Threads

Threads sind Subjekte, die ausführenden Programmcode beschreiben sollen. Ein Thread ist einem bestimmten virtuellen Adressraum zugeordnet, indem sein Programmcode, seine Daten und seinen Stack enthalten ist. Diese Zuordnung findet über einen Prozess statt, bei dem der Thread Mitglied ist.

Grundsätzlich besitzt ein Thread zwei Stacks: einen User-Mode-Stack der von dem Thread normalerweise während der Ausführung im Benutzermodus verwendet wird und einen Kernel-Mode-Stack, der immer dann verwendet wird, wenn dem Thread die CPU-Zeit entzogen worden ist und sein Arbeitszustand vom Kernel gespeichert werden muss oder wenn der Thread in den Kernel-Modus wechselt, um dort einen Systemaufruf auszuführen.

Jeder Thread besitzt eine statische Priorität, die vom Scheduler verwendet wird, um festzulegen wie viel Zeit (effektive Priorität) der Thread pro Scheduling-Durchlauf zur Ausführung zugewiesen

wird, ehe er wieder verdrängt und durch einen anderen ersetzt wird. Dies ist genauer im Kapitel zum Scheduling-Algorithmus erklärt.

Grundsätzlich kann ein Thread sich in einem von vier Betriebsmodi befinden:

- Rechenbereit - Der Thread wartet auf Wiedererhalten der CPU-Zeit
- Rechnend - Der Thread wird auf einer CPU derzeit ausgeführt
- Wartend - Der Thread wartet auf eintreffende Nachrichten oder auf die Empfangsbereitschaft eines anderen Threads

Um allgemein Schutzmaßnahmen zu realisieren kann ein Thread die Kommunikation mit ihm auf bestimmte Prozesse oder Threads einschränken.

Grundsätzlich kann ein Thread nur durch einen Thread des eigenen Prozesses beendet werden. Zusätzlich können Threads die einem sog. *root*-Prozess angehören einen fremden Thread beenden. Ein Thread kann sich nicht selbst beenden, da normalerweise für die Beendigung eines Threads Datenstrukturen vernichtet werden müssen, die der Thread zum Betrieb auch während seiner Beendigung benötigt[2].

1.2.4 Prozesse

Prozesse sind eine Kombination eines virtuellen Adressraums und eines oder mehrerer Threads, die bei ihm Mitglied sind. Mit Hilfe eines Prozesses, können diese Threads als eine Einheit auftreten. Ein Prozess kann in den *root*-Modus gewechselt werden, wodurch dessen Threads bestimmte geschützte Systemaufrufe ausführen kann. Dieser Wechsel wiederum kann nur durch Prozesse durchgeführt werden, die sich selbst im *root*-Modus befinden.

Ein Prozess hat einen sog. *controller thread*, der immer dann verwendet wird, wenn bei einer Thread-spezifischen Aktion nicht die SID eines Threads, sondern nue die eines Prozesses bekannt ist. In Client/Server-Systemen kann dieser Thread die Funktion des Anmelde-Threads des Servers übernehmen, da Clients dann nur noch die Prozess-SID des Servers wissen müssen und die SID des *controller threads* mit Hilfe der Prozesstabelle in Erfahrung bringen können.

Erstellt ein Therad einen neuen Prozess, so darf der Thread den (dann noch leeren) virtuellen Adressraum des neuen Prozesses mit neuen Seitenrahmen anfüllen.

Ein Prozess existiert grundsätzlich nur solange, solange er einen Thread besitzt. Werden alle seiner Threads beendet, so wird auch er beendet. Ein Prozess kann aber in den sog. „Zombie-Modus“ (oder auch „Defunct-Modus“ genannt) gesetzt werden. In diesem Zustand können keine Threads von ihm mehr ausgeführt werden und sein Speicher kann freigegeben werden. Dieser Zwischenzustand ist erforderlich, da in vielen Fällen ein bestimmter Prozess gezielt unterbrochen werden können soll.

1.2.5 Prozessgruppen

Als Übergruppe von Prozessen können Prozessgruppen auftreten. Der Kernel verfeinert das Konzept der Prozessgruppen nicht weiter - sie können für die Implementierung von Benutzer/Gruppen-Systemen im Benutzermodus verwendet werden. Nur die zwei vordefinierten Prozessgruppen können bei der Kommunikationsberechtigung von Threads eingesetzt werden:

- Die *everybody* (`0x04FFFFFF`) Gruppe, bei der jeder Prozess automatisch Mitglied ist (es sei denn, er hat sich ausgeschlossen) und somit immer dann verwendet wird, wenn Threads von allen Threads des Systems Nachrichten annehmen wollen.

[2] Die HydrixOS-API wird jedoch diese Einschränkung mit Hilfe sog. Thread-Pakete verbergen, so dass aus Sicht eines Programmierers ein Thread sich auch selbst beenden kann.

- Die *root* (`0x04000000`) Gruppe. Diese Gruppe repräsentiert alle Prozesse, die sich im *root*-Modus befinden - ein Thread kann somit den Zugriff auf sich auf diese Gruppe beschränken. Prozesse im *root*-Modus haben folgende Rechte:

  – Geschützte Hardwarespeicherbereiche in den virtuellen Adressraum einbauen
  – Auf I/O-Ports zugreifen (transferierbar auf nicht-root)
  – IRQs behandeln (transferierbar auf nicht-root)
  – Prozesse und Threads killen, die nicht zum eigenen Prozess gehören
  – Threads, die nicht zum eigenen Prozess gehören einfrieren / aufwecken
  – Zum Paging-Dämon werden (einmalig möglich)
  – Exceptions, Systemaufrufe und Page-Faults anderer Threads zu sich umleiten
  – Exceptions, Systemaufrufe und Page-Faults in anderem Namen ausführen
  – Register anderer Prozesse auslesen oder ändern
  – Speicherbereichsoperationen anderer Threads aus der Ferne erlauben und ausführen
  – Speicherbereiche anderer virtueller Adressräume freigeben

  Somit besitzen Prozesse der root-Gruppe sehr weitgehende Rechte. Wichtig ist, dass sie einige Rechte auch an Prozesse anderer Gruppen weitergeben können, so dass z.B. Gerätetreibern icht zwingend Mitglieder der Root-Gruppe sein müssen, was die Sicherheit des Systems wesentlich verbessern kann.

1.2.6 Eindeutige IDs

Die SID eines Threads oder eines Prozesses kann in kurzer Zeit mehrfach verwendet werden. Oft ist es jedoch aus Sicherheitsgründen erforderlich, dass ein Prozess eindeutig über lange Zeit hinweg identifiziert werden kann. Aus diesem Grund verfügt jeder Prozess und jeder Thread in seinem Deskriptor über eine eindeutige ID. Die Wiederholung dieser Nummer für einen Prozess oder Thread mit derselben SID ist relativ unwahrscheinlich. Soll also ein Prozess oder Thread eindeutig identifiziet werden, so kann dies zusammen mit dessen SID geschehen: Ein Thread übergibt beispielsweise zu seiner Identifikation einem Server seine SID und seine eindeutige ID. Ist mittlerweile ein neuer Thread mit gleicher SID entstanden, so wird dieser in seinem Deskriptor eine andere eindeutige ID vermerkt haben. Der Server kann somit also feststellen, dass es sich bei diesem Client-Thread trotz identischer SID um einen anderen Thread handelt. Die eindeutige ID lässt sich nicht von Außerhalb manipulieren.

Kapitel 2

Die IPC

2.1 Das grundlende Konzept

Die meisten modernen Mikrokernelsysteme verwenden für die Interprozess-Kommunikation *Message passing*, d.h. das Versenden von Nachrichten zwischen Prozessen. Einige Mikrokernel - wie Mach - bieten dabei sehr komplizierte, asynchrone - d.h. auf Nachrichtenwarteschlangen basierende - Nachrichtensysteme an, andere wiederum - wie L4 - verwenden nur noch sehr kompakte synchrone Nachrichtensysteme, bei denen sehr kurze Nachrichten (evtl. in Kombination mit Speicherverwaltungsbefehlen) versendet werden können.

Das Problem der asynchronen, auf Nachritenwarteschlangen basierenden Nachrichtensysteme ist in erster Linie die niedrige Performance und die Tatsache, dass dabei der Kernel weiterhin große Komplexität besitzt, was letztlich dem Mikrokernelansatz wiederspricht. Andere Systeme, die einen Nachrichtentransfer sehr kurzer Nachrichten anbieten, sind aber meist genauso problematisch. Sie sind zwar sehr effizient und haben bereits eine sehr reduzierte Komplexität - dennoch müssen meist größere Datenmengen ohnehin über gemeinsam genutzte Speicherbereiche transferiert werden, da ansonsten größere Datenmengen nur mit sehr hohen Transferzeiten übertragen werden könnten.

Daher stellt sich die Frage, ob man überhaupt im Kernel ein Nachrichtensystem als solches anbieten sollte, oder ob es nicht genügt, dass der Kernel sehr einfache Dienste zur gemeinsamen Speichernutzung und zur Synchronisierung von Threads anbietet. Der eigentliche Datentransfer über die Speicherseiten kann im Benutzermodus erledigt werden. Dies reduziert die Komplexität des Kernels und erhöht die Systemperformance, da die Daten ohne Kontext-Wechsel übertragen werden können.

Das IPC-Konzept des *hymk* sieht also vor, dass Threads unterschiedlicher Prozesse nur noch über gemeinsam genutzte Speicherbereiche kommunizieren, die über den Kernel eingerichtet werden können und sich mit Hilfe des Kernels synchronisieren. Dadurch soll eine höhere Performance des Systems und eine niedrigere Komplexität des Kernels erreicht werden.

2.2 Gemeinsame Speichernutzung

Für die gemeinsame Speichernutzung sind vier Operationen vorgesehen. Diese Operationen heißen *allow*, *map*, *unmap*, *move*.

2.2.1 allow

allow(SID_src, SID_dest, dest_start, dest_size, operation)

Die Operation *allow* dient dazu festzulegen, von welcher SID Operationen zur gemeinsamen Speichernutzung auf diesen Thread (bzw. dessen virtuellen Adressraum) angewendet werden dürfen (Pa-

rameter *SID_src*). Trotz der Tatsache, dass ein virtuelle Adressraum eigentlich die Sache eines bestimmten Prozesses ist, wird die Zugriffserlaubnis über die Threads gesteuert. Dies wird deshalb gemacht, um einen aufwendigeren, prozess-basierten Zugriffskontrollmechanismus zu vermeiden, der in der Regel überflüssig ist, da die Kommunikation in Client/Server-Systemen ohnehin unmittelbar zwischen dem Client und einem nur für diesen Client erstellen Server-Thread abläuft und es somit völlig ausreicht, dass der Server-Thread (bzw. der Client-Thread der ohnehin nur auf einen Server gleichzeitig zugreifen kann) den Zugriff ohne prozessweite Zugriffslisten steuert.

Die SID, die als Parameter übergeben werden kann, kann die SID eines bestimmten Threads, eines bestimmten Prozesses oder einer der beiden vom Kernel definierten Prozessgruppen (*root* und *everybody*) sein. Sie kann aber auch einem Platzhalter entsprechen. Wird der *null*-Platzhalter verwendet, ist der Zugriff komplett gesperrt. Der *kernel*-Platzhalter ist überflüssig, da der Kernel weder Speicherseiten mit einem Prozess gemeinsam nutzt und wenn er es jemals würde, dies auch ohne Erlaubnis machen könnte.

Root-Threads haben das Recht die allow-Operation auf andere Threads anzuweden, die gar nicht zu ihrem Prozess gehören (Parameter *SID_dest*).

Die Allow-Operation gibt zusätzlich an, auf welche Zieladressen die Speicherverwaltungsoperation angewendet werden soll (Parameter *dest_start* und *dest_size*). Ferner kann festgelegt werden, welche Operationen alle überhaupt zulässig sind (Parameter *operation*).

Der Aufruf *allow* wird automatisch beim Erstellen eines neuen Prozesses aufgerufen, so dass der erzeugende Thread Speicherinhalte in den neuen Prozess hineinverschieben kann. Abgesehen von dieser Ausnahme werden sonst nach dem Start eines neuen Threads die von *allow* zu definierenden Einstellungen so gesetzt, dass kein anderer Thread Speicherverwaltugnsoperationen ausführen kann.

2.2.2 map

map(SID, src_start, src_size, flags)

Die Operation *map* stellt einen gemeinsamen Speicherbereich des eigenen virtuellen Adressraums dem virtuellen Adressraum eines anderen Threads (Parameter *SID*) zur Verfügung. Der Speicherbereich wird durch eine Startadresse und eine Größe definiert (Parameter *src_start* und *src_size*). Zusätzlich können Flags definiert werden, die die Zugriffsrechte auf den Bereich festlegen (Nur-Lesen, Lesen/Schreiben, Lesen/Schreiben/Ausführen), sofern diese durch die Hardware-Plattform unterstützt werden. Außerdem kann festgelegt werden, dass die Seiten für das Copy-On-Write-Verfahren markiert werden sollen, so dass effektiv kein gemeinsamer Speicherbereich, sondern tatsächlich zwei getrennte Speicherbereiche entstehen.

Der Zugriff auf den anderen virtuellen Adressraum wird durch eine vorangegangene *allow*-Operation der Gegenseite bestimmt. Der map-Aufruf kann auch auf den eigenen Prozess angewendet werden, erfordert aber auch eine *allow*-Operation, um die Zieladresse festzulegen.

2.2.3 unmap

unmap(SID, dest_start, dest_size, flags)

Entfernt einen Speicherbereich aus einen Zieladressraum, der zu einem Zielthread gehört (Parameter *SID*). Diese Operation betrifft nicht zwingend gemeinsam genutzte Speicherbereiche, sondern kann auch andere Speicherbereiche betreffen - somit können z.B. auch normale Speicherbereiche freigegeben werden. Mit Hilfe bestimmter Flags können auch nur Zugriffsrechte reduziert werden. Root-Prozesse können diesen Aufruf auch ohne ein vorausgegangenes *allow* der Gegenseite ausführen.

Der *unmap*-Aufruf kann ohne Aufruf von *allow* auch auf den eigenen Prozess angewendet werden, um lokalen Speicher freizugeben oder Mappings aus dem eigenen Adressraum zu entfernen.

2.2.4 move

move(SID, src_start, src_start, flags)

Dieser Aufruf ist eine Mischung aus einem *map* auf einen anderen Adressraum und einem *unmap* auf den eigenen Adressraum mit dem Ziel einen Speicherbereich zu einem anderen Adressraum zu verschieben. Der andere virtuelle Adressraum, repräsentiert durch den Thread, dessen SID definiert ist (Parameter *SID*) muss daher zuvor mit einem *allow* freigegeben worden sein (der eigene Adressraum natürlich nicht). Die zu verschiebenden Speicherbereiche werden durch eine Startadresse und eine Bereichsgröße definiert (Parameter *src_start* und *src_size*). Ferner können auch hier Zugriffsflags definiert werden.

Ist der Zielbereich kleiner als der Quellbereich, werden überschüssige Speicherseiten freigegeben.

2.3 Synchronisation von Threads

Für die gegenseitige Synchronisation von Threads gibt es einen Systemaufruf, der von beiden zu synchronisierenden Seiten verwendet wird:

2.3.1 sync

sync(SID, timeout, resync);

Synchronisiert zwei Threads innerhalb eines bestimmten (oder unendlichen) Timeouts miteinander - d.h. der Aufrufer wartet, bis ein gewählter anderer Thread (definiert durch den Parameter *SID*) ebenfalls *sync* mit seiner SID aufruft.

Das Warten geschieht weitgehend passiv, so dass durch den Wartevorgang keine oder relativ wenig CPU-Zeit verbraucht wird. Der Kernel kann zur Umsetzung interne Warteschleifen einsetzen. Gewartet wird solange, bis die Gegenseite die Synchronisierung gestattet - d.h. auch wenn die Gegenseite sich derzeit auf eingehende Synchronisationen wartet, aber die Synchronisation mit dem Aufrufer verbietet, wartet der Aufrufer, bis sich dieser Zustand der Gegenseite ändert oder das Timeout abläuft.

Die Gegenseite, auf die ein Thread wartet, kann

- ein bestimmter Thread (definiert durch dessen Thread-SID) sein
- ein beliebiger Thread eines bestimmten Prozesses (definiert durch die Prozess-SID) sein
- ein beliebiger Thread aller Prozesse sein, die sich im *root*-Modus befinden (definiert durch die *root*-SID)
- ein beliebiger Thread (definiert durch die *everybody*-SID) sein

Wichtig ist hierbei, dass bei einer Synchronisierung grundsätzlich mindestens einer der beiden Seiten eine konkrete Thread-SID angeben muss, da sonst das Ziel nicht genau spezifiziert ist.
Um unnötige Wechsel zwischen Kernel- und Benutzermodus zu vermeiden, kann ein Thread eine automatische Wiederholung des selben Synchronisationsvorgangs bei erfolgreicher Synchronisation anordnen. In einem Client-Server-System ist es z.B. oft der Fall, dass der Client sich zuerst mit dem Worker-Thread des Servers synchronisiert, um diesen zu signalisieren, dass ein neuer Auftrag ansteht und anschließend auf eingehende Synchronisation des Worker-Threads wartet, damit dieser ihm die Fertigstellung des Auftrags signalisiert. Durch die automatische Resynchronisierung kann hier ein überflüssiger Wechsel zwischen Kernel- und Benutzeradressraum vermieden werden.

2.3.2 Weitere Synchronisationsmöglichkeiten

freeze_subject und awake_subject

Es gibt auch weitere Synchronisationsmöglichkeiten über die Operationen *freeze_subject* und *awake_subject*, mit deren Hilfe Threads von außen angehalten oder fortgesetzt weden können. Diese Operation kann von allen Root-Prozessen oder aber auch von Threads des gleichen Prozesses untereinander angewandt werden.

Diese Aufrufe sind auf Threads, die sich im Kernel-Modus befinden möglicherwiese nur bedingt anwendbar (implementationsabhängig).

Kapitel 3

Kontrolloperationen

3.1 Allgemeine Fernsteuerung von Threads

Der *hymk* bietet eien Reihe von Funktionen an, mit deren Hilfe andere Threads ferngesteuert werden können. Diese Funktionen sind grundsätzlich auf root beschränkt. Diese allgemeine Fernsteuerung wurde aus verschiedenen Gründen eingebaut - einerseits um schlichtweg einen Debugging-Mechanismus zu ermöglichen, anderseits aber auch um im Benutzermodus Virtualisierungs- und Übersetzungsmechanismen zu ermöglichen, die für die von HydrixOS angebotene Architekturtransparenz unbedingt erforderlich sind.

3.1.1 Kontrolle der Register eines Threads

Grundsätzlich gehört zu dieser Fernsteuerungsmöglichkeit das Auslesen und Ändern der Register eines Threads. Sie schließen den Programm- und Stackzeiger mit ein. Diese Operationen sind von ihrer Schnittstelle selbstverständlich immer sehr plattformspezifisch. Grundsätzlich können sie nur dann ausgeführt werden, wenn der zu manipulierende Thread mit einem *freeze_subject*-Aufruf zuvor angehalten wurde (hierbei muss der Thread explizit angehalten sein, nicht nur dessen Prozess).

3.1.2 Kontrolle der Softwareinterrupts

Das wichtigste Kontrollfeature ist die Kontrolle von Softwareinterrupts. Der kontrollierende Thread kann sich bei der Ausführung von Softwareinterrupts zwischenschalten, so dass diese nicht vom Kernel, sondern von dem kontrollierenden Thread behandelt werden können. Dabei wird die Nummer des Softwareinterupts mitgeteilt und der ausführende Thread mit *freeze_subject* temporär gestoppt. Softwareinterrupts schließen unter HydrixOS Systemaufrufe, Exceptions und Page Faults ein. Prinzipiell kann nur ein Thread gleichzeitig sich beim gleichen Thread zwischenschalten.

Auf der anderen Seite können aber auch Softwareinterrupts vorgetäuscht werden. Somit kann ein Thread von außen einerseits einen Page-Fault oder eine Exception in dem betroffenen Thread auslösen, aber besonders im Namen eines anderen Threads einen Systemaufruf ausführen.

In beiden Fällen werden die Softwareinterrupts mit den Registerwerten des kontrollierten Threads ausgeführt, bzw. von dort die Parameter bei einer Zwischenschaltung bezogen, um maximale Transparenz zu erreichen.

3.2 Spezielle Fernsteuerung des Paging Dämons

3.2.1 Behandlung von Page Faults

Eine spezielle Fernsteuerung erfolgt immer durch den Paging Dämon. Dieser Dienst wird vom Kernel immer aufgerufen, wenn ein *Page Fault* nicht durch eine Zwischenschaltung (siehe 3.1.2) behandelt wurde. Um diese *Page Faults* entgegenzunehmen, geht der Paging Dämon mittels eines sync auf den Kernel-Platzhalter in einen Wartemodus. Tritt ein *Page Fault* auf, wird der Paging Dämon reaktiviert und erhält als Rückgabewert die Thread-SID des Threads, der den Fault ausgelöst hat. Mit dieser Thread-SID kann er einen aus dem Thread-Deskriptor des betroffenen Threads auslesen. Dieser enthält hierfür die folgenden Informationen:

- Betroffene Speicheradresse
- Inhalt des Page-Deskriptors (Plattformabhängig; fehlt die notwendige Seitentabelle, wird der Deskriptor der Seitentabelle übermittelt)
- Abgebrochene Operation (Lesen, Schreiben, Freigeben der Speicherseite)
- Weitere Daten zur aufgetretenen Exception, wie z.B. der plattformspezifische Fehlercode (siehe Exception-Handling)

Der Paging-Dämon kann dann eine *allow*-Operation im Namen des betroffenen Threads ausführen (allerdings muss er den alten *allow*-Zustand danach auch wiederherstellen) und ggf. Pages von einem exterenen Medium zurückmappen oder aber eine Exception (außer einem Page-Fault) im Namen des Threads ausführen, falls keine Page zur Verfügung steht.

Solange der Paging-Dämon nicht mit sync() auf neue Page-Faults wartet, bleiben diejenigen Threads im Kernel-Mode stehen, die den Page Fault erzeugt haben (der Teil des Thread-Deskriptors mit den Informationen zum Page-Fault ändert sich auch bis dahin nicht). Findet die Synchronisierung statt, wird der auslösende Thread während des Page-Faults mit *freeze_subject* eingefroren. Der Paging-Dämon muss ihn also anschließend wieder mit *awake_subject* aufwecken.

3.2.2 Behandlung von Exceptions

Löst ein Thread eine Exception (*Ausnahmefehler*) aus, so wird ebenfalls der Paging-Dämon informiert. In diesem Fall können folgende Informationen aus dem Deskriptor des betroffenen Threads entnommen werden:

- Plattformunabhängige Fehlernummer
- Ausgelöste Exception (Plattformabhängig)
- Fehlercode (Plattformabhängig)

Der Thread, der diese Exception ausgelöst hat, wurde mit *freeze_subject* zuvor bereits eingefroren. Je nach Implementierung kann der Paging-Dämon dann diese Informationen weiterleiten, die Ausführung des Threads mit *awake_subject* wieder fortsetzen (sofern die Ursache der Exception beseitigt wurde) oder den betroffenen Thread einfach beenden.

Trat die Exception auf, bevor ein Paging-Dämon installiert werden konnte oder trat die Exception im Paging-Dämon selbst auf, so wird das System angehalten.

3.2.3 Installation des Paging-Dämons

Der Paging-Dämon meldet sich beim Start des Systems über den Systemaufruf *set_paged* beim Kernel an. Ist er einmal angemeldet, kann kein anderer Thread sich mehr als Paging-Dämon anmelden, bis dieser Thread beendet wurde.

3.2.4 Anmerkungen zur Implementierung

Ein Paging-Dämon, der das Ziel hat, Speicherseiten auf ein externes Medium auszulagern, kann mit den derzeitigen Mitteln noch kaum oder zumindest sehr schwierig implementiert werden. Es wird sicherlich weitere Erweiterung der Systemarchitektur erfordern, um diese Fähigkeit des Systems effizient auszubauen. Auf Grund der hohen RAM-Speicherkapazitäten und des Bedarfs an einem unkomplizierten Betriebssystemkern wurde der Aspekt der Seitenauslagerung vorerst noch ausgelassen.

Kapitel 4

System- und Hardwareverwaltung

4.1 IRQs

4.1.1 Funktionsweise von IRQs

Auf nahezu jeder Plattform können auf Grund angeschlossener, externer Geräte bestimmte Ereignisse asynchron zur normalen Programmausführung auftreten und müssen dementsprechend behandelt werden. Wird beispielsweise eine Taste gedrückt, so muss u.U. die aktuelle Programmausführung unterbrochen werden, damit die Eingaben an einen Thread weitergeleitet werden können, der Tastatureingaben erwartet. Diese Art der Behandlung externer Ereignisse wird durch das Konzept der IRQs gelöst.

Ein externes Gerät meldet der CPU über einen bestimmten, hardwaremäßigen Weg, dass ein externes Ereignis eingetreten ist und stellt somit eine Anfrage auf eine Unterbrechung der aktuellen Ausführung (*interrupt request* = IRQ). Die CPU prüft, ob dieser IRQ vom laufenden Betriebssystem angenommen wird oder nicht. Wird er angenommen, so unterbricht die CPU die aktuelle Ausführung und sichert ihren Betriebszustand auf eine plattformspezifische Weise (meist auf den Kernel-Stack des aktuellen Threads). Anschließend führt sie eine Behandlungsroutine aus, die das Betriebssystem für diesen IRQ angegeben hat. Dieses leitet den IRQ an den jeweiligen Treiber weiter, der dann das Ereignis behandelt (z.B. Daten aus Gerätepuffern lädt etc.). Anschließend erklärt der Treiber den IRQ für behandelt und das System stellt den Zustand vor dem IRQ wieder her (oder führt ggf. einen Kontext-Wechsel durch).

4.1.2 Behandlung von IRQs im *hymk*

Unter dem *hymk* sind zwei Phasen der IRQ-Behandlung zu unterscheiden: Die *Low-Level*-Behandlung der IRQs und die *High-Level*-Behandlung der IRQs. Die Low-Level-Behandlung ist ein sehr plattform- und implementationsabhängiger Teil der IRQ-Behandlung. Die High-Level-Behandlung hingegen wurde soweit verallgemeinert, dass sie nahezu plattformunabhängig sein sollte.

Im Unterschied zu anderen Mikrokernel-Systemen wurde die IRQ-Behandlung unter HydrixOS nicht über den generischen IPC-Mechanismus implementiert, sondern durch einen speziellen Mechanismus. Der Grund ist folgender: die Behandlung eines IRQs kann mit einer plattformspezifischen Anpassung wesentlich performanter ausgeführt werden. Zudem verwenden viele Systeme, die nach außen hin den gleichen IPC-Mechanismus für die IRQ-Behandlung einsetzten, intern doch einen spezifischen Mechanismus, was nur dazu führt, dass die Anzahl der Systemaufrufe, aber nicht deren interne Komplexität reduziert wird.

Bei der High-Level-Behandlung eines IRQs ruft ein Thread, der dazu die Erlaubnis hat, die Routine *recv_irq* auf. Diese Routine prüft zuerst, ob bereits ein anderer Thread auf den gleichen IRQ

wartet. Ist dem so, wird der Vorgang abgebrochen. Andernfalls trägt die Routine den Thread zur IRQ-Behandlung ein und legt ihn in einen passiven Wartemodus. Tritt der IRQ auf, wird der aktuelle Thread verdrängt und der IRQ-Thread aus dem Wartemodus erweckt (dieser Thread hat eine sehr hohe Priorität, wodurch er alle anderen Threads verdrängt werden). Anschließend kann er die IRQ-Behandlung ausführen. Ist er fertig, kann er den selben IRQ erneut mit *recv_irq* behandeln. Sollte er keine weiteren IRQs mehr behandeln wollen, so hat er dennoch *recv_irq* mit einem plattformspezifischen Wert auszuführen, der einen ungültigen IRQ symbolisiert, um den zuletzt behandelten IRQ wieder freizugeben. In beiden Fällen löst der Aufruf von *recv_irq* während einer IRQ-Behandlung interne Operationen aus, die hardwaremäßig erforderlich sind, um den IRQ als behandelt zu erklären.

Grundsätzlich ist während einer IRQ-Behandlung die Behandlung weiterer IRQs gleichzeitig möglich, da während der IRQ-Behandlung nicht alle, sondern nur der zu behandelnde IRQ maskiert wird. Daher können IRQ-Behandlungsroutinen prinzipiell auch blockierende Operationen - wie sync - ausführen. Die Priorisierung der IRQs ist bei den meisten Plattformen durch die Hardware geregelt.

4.2 I/O-Speicher

4.2.1 Speicherbasiertes I/O

Bei vielen Plattformen werden externe Geräte über Speicherbereiche angesteuert, die in den physikalischen Adressraum der CPU eingeblendet wurden. Diese Speicherbereiche können meist nur mit eingeschränkten Cache-Regelungen verwendet werden, so dass sie nicht direkt als normale Seitenrahmen verwendet werden dürfen. Stattdessen ist ein spezieller Reservierungsmechanismus erforderlich, der zudem eine mehrfache Resevierung gleicher Speicherbereiche erlaubt.

Der Reservierungsmechanismus des *hymk* erlaubt für Root-Prozesse die Aufnahme von I/O-Adressbereichen in deren eigenen virtuellen Adressraum über den Systemaufruf *io_alloc*. Falls erforderlich können Root-Prozesse mit *map* diese Adressräume dann an Nicht-root-Prozesse weitergeben, so dass Treiberimplementierungen zur Verbesserung der Systemsicherheit in weniger privilegierten Prozessen durchgeführt werden können.

4.2.2 Portbasiertes I/O

Einige Plattformen (in erster Linie Intels x86-Plattform) bieten neben dem Speicherbasierten I/O auch noch einen weiteren Adressraum von I/O-Ports an. Dieser Zugriff wird meist über eigene Instruktionen realisiert und kann für einzelne Prozesse meist sehr detailiert eingeschränkt werden. Der *hymk* erlaubt grundsätzlich allen Root-Prozessen den Zugriff auf alle I/O-Ports. Root-Prozesse können aber einzelnen Nicht-root-Prozessen den Zugriff auf die I/O-Ports zusätzlich gestatten.

4.3 I/O-Sicherheit

Wie bereits in den vorherigen Kapiteln erwähnt, ist I/O meist nur auf root-Prozesse beschränkt. Jedoch können Root-Prozesse bestimmte I/O-Rechte an Nicht-root-Prozesse weiterleiten. Dies ist recht sinnvoll, da dadurch Treiber nicht mit vollen Root-Privilegien arbeiten müssen, sondern auch unter beschränkten und kontrollierbaren Privilegien arbeiten können. Die Privilegienkontrolle erfolgt über den Systemaufruf *io_allow*. Mit diesem lassen sich folgende Privilegien regeln:

- Zugriff auf die Portbasiert-I/O (falls vorhanden)
- Behandlung von IRQs

Die Weitergabe von IO-Adressbereichen kann über das mappen solcher Adressbereiche durch root-Prozesse erfolgen. Somit ist dort eine feinkörnigere Steuerung möglich.

4.4 Info-Pages

Bei vielen Systemen werden aktuelle Statusinformationen meist durch Systemaufrufe zugänglich gemacht. Da dies jedoch meist einen weiteren Kontext-Wechsel bedeutet und zudem die dabei erledigten Aufgaben meistens lediglich auf eine bestimmte Systemtabelle zugreifen, verzichtet der *hymk* auf solche Aufrufe. Statt dessen blendet er am Ende des Kernel-Adressraums die sogenannten Info-Pages ein (die genaue Adresse und die größe dieses Areals sind Plattform bedingt). Dieser Bereich ist für Programme im User-Mode lesbar und für den Kernel auch beschreibbar.

Der exakte Aufbau der Inhalte der Info-Pages und die Position von diesen ist plattformspezifisch - daher müssen API-Bibliotheken Zugriffsfunktionen anbieten, die den Zugriff auf die eigentlichen Informationen plattformunabhängig ermöglichen. Grundsätztlich setzt sich dieser Bereich immer aus den folgenden Teilbereichen zusammen:

1. Der Hauptinfopage
2. Der Prozesstabelle
3. Der Threadtabelle

4.4.1 Die Hauptinfopage

Die Hauptinfopage enthält Informationen über den aktuellen Systemzustand. Diese sind auf allen Plattformen mindestens die folgenden:

- Die SID des aktuellen Prozesses
- Die SID des aktuellen Threads
- Einen Zeiger auf den Prozesstabelleneintrag des aktuellen Prozesses
- Einen Zeiger auf den Threadtabelleneintrag des aktuellen Threads
- Die Versionsnummer des Kernels
- SID des Paging-Dämons
- Der aktuelle RTC-Zähler
- Der Idendifikationscode für die CPU (z.B. 0x80386)
- Die Größe einer Speicherseite auf dieser Plattform
- Plattformspezifische Daten, die über die CPU (u.ä.) nähere Auskunft geben
- Anzahl von Seiten, die maximal in einem Systemaufruf (z.B. alloc_pages, io_alloc, map)

Der genaue Aufbau der Infopage für die jeweilige Kernel-Implementierung ist später im Handbuch beschrieben.

4.4.2 Die Prozesstabelle

Die Prozesstabelle enthält alle Informationen, die der Kernel über den jeweiligen Prozess speichert. Es handelt sich dabei um die tatsächlichen Prozessdeskriptoren, die der Kernel auch intern verwendet (der Kernel benutzt ebenfalls die Info-Pages um diese Deskriptoren zu manipulieren, da dies einen schnellen Zugriff über Arrays gestattet, ohne physischen Speicher zu verschwenden). Diese Tabelle entählt für jede im System möglich Prozess-SID (auf x86-Systemen sind dies 4096) einen Deskriptor. Ist zu einer Prozess-SID kein Deskriptor verfügbar, wird ein leerer Deskriptor zur Verfügung gestellt (es handelt sich dabei immer um den gleichen Seitenrahmen). Um Speicherplatz zu sparen, ist diese Tabelle ein Mapping von Deskriptoren, die in irgendwelchen Seitenrahmen gespeichert wurden, welche im physikalischen Speicher keine zusammenhängende Tabelle bilden.

Normalerweise besteht ein Eintrag aus mehreren Teilen: in einem ist der tatsächliche Deskriptor des Prozesses enthalten und im anderen eine Speicherverwaltungstabelle des Kernels enthalten, deren Struktur implementations-abhängig ist. Meistens sind diese Tabelleneinträge über mehrere vollständige Speicherseiten verteilt. Der genaue Aufbau der Prozesstabelle ist von der jeweiligen Kernel-Implementierung abhängig. Dennoch sollen von unterschiedlichen Kernel-Implementierungen im wesentlichen der gleiche Bestand an Grundinformationen angeboten werden.

Die Details über den Aufbau der Info-Pages können der Dokumentation der jeweiligen Implementierung entnommen werden. Es empfiehlt sich jedoch die interne Struktur der Prozessdeskriptoren über eine API-Bibliothek zu kapseln.

4.4.3 Die Threadtabelle

Die Threadtabelle ist der Prozesstabelle sehr ähnlich. Auch sie enthält alle Informationen, die der Kernel über den jeweiligen Thread speichert. Auch handelt es sich hier um die tatsächlichen Threaddeskriptoren, die der Kernel auch intern verwendet. Diese Tabelle enthält für jede im System möglich Thread-SID (auf x86-Systemen sind dies 4096) einen Deskriptor. Ist zu einer Thread-SID kein Deskriptor verfügbar, wird auch hier immer der gleiche leere Deskriptor zur Verfügung gestellt. Um Speicherplatz zu sparen, ist auch diese Tabelle ein Mapping von Deskriptoren, die in irgendwelchen Seitenrahmen gespeichert wurden, welche auch in diesem Fall im physikalischen Speicher keine zusammenhängende Tabelle bilden. Normalerweise besteht ein Eintrag aus drei Teilen: in einem ist der Deskriptor des Threads enthalten und im zweiten der *thread local storage* des Threads. Meistens sind diese Einträge über mehrere vollständige Speicherseiten verteilt.

Auch der genaue Aufbau der Threadtabelle ist von der jeweiligen Kernel-Implementierung stark abhängig und können der Dokumentation der jeweiligen Implementierung entnommen werden. Auch hier empfiehlt sich eine Kapselung durch API-Funktionen.

4.5 Der Thread local storage

Neben den Info-Pages gibt es noch den *thread local storage*. Dabei handelt es sich um eine Speicherseite, die an der letzten oder vorletzten Adresse des Benutzeradressraums eines virtuellen Adressraums liegt. Der zu dieser speziellen Seite zugeordnete Seitenrahmen wechselt dabei mit jedem Thread, so dass jeder Thread in diesem Adressbereich seinen eigenen, privaten Speicherplatz hat, auf den nur er zugreifen kann und der an einer festgelegten Adresse liegt. Der Seitenrahmen zu diesem Speicherbereich wird automatisch bei der Erzeugung des Threads angelegt und beim Beenden des Threads freigegeben.

In diesen *thread local storage* kann er z.B. Zeiger auf API-Variablen legen, die Thread-spezifisch sind und auf die immer sehr schnell zugegriffen werden können muss (z.B. der aktuelle API-Fehlerzustand).

Kapitel 5

Programmausführung

5.1 Der Scheduler

5.1.1 Organisation des Schedulers

Wie jedes moderene Betriebssystem unterstützt HydrixOS natürlich preemptives Multitasking (bzw. Multithreading). Dabei unterstützt der *hymk* derzeit eine Prioritätenskala von 40 Prioritäten - von der niedrigsten Priorität 0, bis zur höchsten Priorität 40. Aus dieser statischen Priorität wird, wenn ein Thread aktiviert wird (z.B. nachdem er erstellt wurde oder er einen I/O-Wartezustand verlassen hat), eine effektive Priorität ausgerechnet. Diese effektive Priorität gibt die Länge der Zeit an, in der ein Thread die CPU verwenden darf. Diese effektive Priorität wird dazu mit jedem Uhrenschlag dekrementiert (außer der Thread ist im Kernel-Modus).

Ist jedenfalls die effektive Priorität eines neu aktivierten Threads höher als die des derzeit laufenden Threads, so wird der derzeitige Thread verdrängt und dem höherprioritäre Thread die CPU übergeben. Ist die effektive Priorität eines Threads aufgebraucht, so wird er ebenfalls verdrängt und durch einen anderen (beliebigen) Thread ersetzt.

Dieser Vorgang wird ständig mit allen Threads gemacht, die in der sog. Run-Queue des Schedulers abgelegt werden. In dieser Warteschlange werden grundsätzlich alle rechenbereiten Threads abgelegt, so dass nicht nach erneut rechenbereiten Threads gesucht werden muss. Wird ein neuer Thread mit höherer Priorität als der des aktuellen Threads rechenbereit, so wird er in diese Schlange direkt nach dem aktuellen Thread eingereiht. (Die Warteschlange ist als verkettete Liste organisiert).

5.1.2 Bevorzung interaktiver Prozesse

Grundsätzlich ist der Scheduler auf die Förderung interaktiver Programme ausgerichtet, die immer wieder auf Eingaben oder Ausgabemöglichkeiten warten, dann aber möglichst schnell reagieren müssen. Dies wird über mehrere Verfahren erreicht: Zum einen können zwei Threads, die in gegenseitige Interaktion treten gezielt ihre restliche effektive Priorität an die jeweilige Gegenseite abtreten (d.h. die abgetretene effektive Priorität wird zu der der Gegenseite addiert), die für sie eine Aufgabe erledigen soll (so kann z.B. ein Client beim Aufruf eines Server-Threads seine effektive Priorität an den Server-Thread abgeben, damit dieser die vom Client gestellte Aufgabe schneller erledigen kann).

Auf der anderen Seite werden grundsätzlich Programme gefördert, die auf Eingaben warten, indem beim Betreten des Wartemodus ihre restliche effektive Priorität gesichert wird und später beim Wiedererwachen zur ohnehin zur Verfügung stehenden effektiven Priorität dazuaddiert wird. Dies bringt den Vorteil, dass dieses Programm in einem Zug mehr Aufgaben erledigen kann, aber vorallem erhöht es die Wahrscheinlichkeit, dass ein laufender Thread auf Grund von dessen meist niedrigeren effektiven Priorität durch den aus dem Wartemodus erweckten Thread verdrängt werden kann.

5.1.3 Timeouts

Damit *sync* und *recv_softints*-Operationen, die gesetzte Timeouts besitzen, nicht jedesmal einen vollständigen Kontextwechsel zur Überprüfung des Timeouts erfordern, wird diese Überprüfung beiläufig bei jedem Clock-Tick durchgeführt. Erst wenn das gesetzte Timeout abgelaufen ist, wird der wartende Thread wirklich wieder reaktiviert - andernfalls wird lediglich der Zähler für das Timeout dekrementiert und ein anderer Thread ausgewählt.

5.2 Systemaufrufe

Die Systemaufrufe des *hymk* sind grundsätzlich nicht vollständig portabel entworfen worden. Gerade Parameter, die Speicheradressen beinhalten, verwenden auf den verschiedenen Plattformen unterschiedliche Datentypen. Ebenfalls ist der Mechanismus, der zum Aufruf verwendet werden soll, grundsätzlich plattformabhängig. Das aber, was die einzelnen Systemaufrufe bewirken sollen, sollte auf allen Plattformen identisch sein (abgesehen von evtl. erforderlichen zusätzlichen plattformspezifischen Aufrufen und Detail-Parametern).

Da also die genaue Aufrufmethode und die Parameter in ihren Einzelheiten nicht plattformunabhängig erklärt werden kann, werden in diesem Kapitel nur die allgemein verfügbaren Systemaufrufe kurz erläutert. Ihre detailierte Spezifikation sind in dem für die jeweilige Plattform bestimmten Teil dieses Dokumentes aufgelistet:

Name	**Beschränkungen**	**Kurzbeschreibung**
alloc_pages	Aktueller Thread	Reserviert Speicherseiten und mappt sie in den aktuellen virtuellen Adressraum.
create_thread	Aktueller Thread	Erzeugt einen neuen Thread.
create_process	Aktueller Thread	Erzeugt einen neuen Prozess.
set_controller	Aktueller Prozess	Legt den *controller thread* für den aktuellen Prozess fest.
destroy_subject	Nur Paging-Dämon	Zerstört einen Prozess oder Thread
chg_root	Root;	Wechselt einen Prozess in den Root-Modus oder entfernt ihn daraus.
freeze_subject	Threads des gleichen Prozesses; Root;	Friert einen Thread ein.
awake_subject	Threads des gleichen Prozesses; Root;	Weckt einen mit *freeze_subject* eingefrorenen Thread wieder auf.
yield_thread	Aktueller Thread	Gibt die restliche effektive Priorität an einen anderen Thread (oder keinen anderen) ab.
set_priority	Threads des gleichen Prozesses (nur herabsetzen); Root;	Ändert die aktuelle Priorität. Nur Root darf die Priorität dabei anheben.
allow	Aktueller Thread; Root;	Erlaubt eine Seitenverwaltungsoperation für eine bestimmte SID
map	Nur *allow* oder Root	Etabliert eine gemeinsame Speichernutzung
unmap	Nur *allow* oder Root	Entfernt Speicherseiten aus einem virtuellen Adressraum oder reduziert zumindest die Zugriffsrechte darauf

move	Nur *allow* oder Root	Verschiebt Speicherseiten in einen anderen virtuellen Adressraum
sync	Abhängig von Gegenseite	Synchronisiert zwei Threads
io_allow	Nur Root	Erlaubt bestimmte I/O-Systemaufrufe und I/O-Instruktionen für nicht-Root-Prozesse
io_alloc	Nur Root	Mapt einen I/O-Adressbereich in den aktuellen virtuellen Adressraum
recv_irq	Root / io_allow	Meldet eine IRQ-Behandlung an und wartet auf diese. Führt ebenfalls die Rückkehr aus einer IRQ-Behandlung durch.
recv_softints	Nur Root	Fängt alle Software-Interrupts (Systemaufrufe etc.) eines Threads ab
read_regs	Nur Root	Liest die Registerinhalte eines anderen Threads
write_regs	Nur Root	Schreibt in die Register eines anderen Threads
set_paged	Nur Root	Legt den aktuellen Thread als Paging-Dämon fest

5.3 Fehlerbehandlung

Der Kernel kennt grundsätzlich drei Arten von Fehlern, die unterschiedlich behandelt werden: *Kernel-Mode Exceptions*, *User-Mode Exceptions*, *Systemaufrufsfehler*.

5.3.1 Kernel-Mode-Exceptions

Eine Kernel-Mode-Exception ist ein asynchrones Ereignis, das eintritt, wenn ein Programm im Kernel-Modus Code ausführt, dessen Ausführung auf Grund bestimmter Instruktionsparameter oder auf Grund der Instruktion selbst, von der CPU nicht gestattet werden kann. Ein klassisches Beispiel ist die „Division durch Null“ oder der Zugriff auf eine ungültige Speicherseite. Da der Kernel mit größter Sorgfalt programmiert werden muss, um dem Anspruch des „vertrauenswürdigen Teils“ des Systems gerecht werden zu können, ist eine Kernel-Mode-Exception immer ein Hinweis darauf, dass der Kernel einen Fehler enthält oder fehlerhaft ist, weil er Eingabeparameter eines Systemaufrufs nicht richtig geprüft hat. Seltener könnte auch ein schwerer Defekt der Hardware die Ursache für eine Kernel-Mode-Exception sein. Daher führt eine Kernel-Mode-Exception immer zum Stopp des gesammten Systems mit Ausgabe einiger kurze Debugging-Informationen.

Da der Kernel eine niedrige Komplexität besitzt und auch kaum erweitert werden dürfte, sollten Kernel-Mode-Exceptions nur in anfänglichen Testphasen der Systemimplementierung oder aber bei schweren Hardwaredefekten (z.B. defekten Speichermodulen) auftreten.

Die möglichen Fehlernummern und Fehlercodes für Kernel-Mode-Exceptions sind plattformspezifisch und können aus den Dokumentationen der jeweiligen verwendeten CPU-Sorte bezogen werden.

5.3.2 User-Mode-Exceptions

Eine User-Mode-Exception ist ähnlich einer Kernel-Mode-Exception ebenfalls ein asynchrones Ereignis, das eintritt, wenn ein Programm Code ausführt, den die CPU auf Grund von einem Parameter, einer Speicherreferenz oder der Instruktion selbst als fehlerhaft eingestuft hat. Im Unterschied zu einer Kernel-Mode-Exception handelt es sich jedoch um Code den ein Programm im Benutzermodus ausgeführt hat, was normalerweise auf einen Programmierfehler des ausgeführten Programms zurückzuführen ist.

Im Fall einer User-Mode-Exception wird der fehlerhafte Thread mit *freeze_subject* eingefroren und der Vorfall entweder dem Paging-Dämon oder aber an ein Programm weitergegeben, dass die Softwareinterrupts des fehlerhaften Programms abgegriffen hat. Diese sind dann für die weitere Behandlung der Exception verantwortlich. Die übliche Reaktion ist die Beendigung des fehlerhaften Threads oder der Aufruf eines Debuggers.

Der *hymk* unterscheidet zwischen User-Mode-Exceptions und Page Faults, auch wenn diese auf den meisten Plattformen grundsätzlich unter die Kategorie „Exception“ fallen und beide auch über das Abgreifen der Softwareinterrupts ferngesteuert werden können. Dieser Unterschied wurde eingebaut, um dem Paging-Dämon eine schnellere und plattformunabhängige Unterscheidung von Page Faults und Exceptions zu ermöglichen.

User-Mode-Exceptions sind normalerweise mit einer plattformabhängigen Fehlernummer (repräsentiert durch den ausgelösten Softwareinterrupt) und einen plattformabhängigen Fehlercode verbunden. Zusätzlich liefert der Kernel für den Paging-Dämon (und andere Programme) auch plattformunabhänige Fehlernummern für Usermode-Exceptions:

Name	**Nummer**	**Bedeutung**
`EXC_DIVISION_BY_ZERO`	1	Division durch Null.
`EXC_PROTECTION_FAULT`	2	Speicherzugriffsfehler.
`EXC_INVALID_INSTRUCTION`	3	Ungültige Instruktion.
`EXC_TRAP`	4	Der Haltepunkt eines Debuggers wurde erreicht (Ursachen sind sehr plattformspezifisch).
`EXC_OVERFLOW`	5	Bei einer Rechenoperation wurde erzeugt (Ursachen sind sehr plattformspezifisch).
`EXC_STACK_FAULT`	6	Der Zugriff auf den Stack schlug fehl.
`EXC_INVALID_PAGE`	7	Es wurde auf eine ungültige Speicherseite zugegriffen oder der Paging-Dämon ist nicht verfügbar.
`EXC_SPECIFIC`	8	Es trat eine plattformspezifische Exception auf, für die keine generelle Fehlernummer existiert

5.3.3 Systemaufrufsfehler

Neben Exceptions gibt es auch Fehler, die während der Ausführung eines Systemaufrufs auftreten können und auf Grund ungültiger Parameter oder gesperrter Zugriffsrechte ausgelöst wurden. Diese Fehler werden vom Kernel einfach mit dem Rückgabewert des jeweiligen Systemaufrufs auf eine

plattformspezifische Weise synchron zurückgegeben. Derzeit sind folgende Fehlernummern für die Systemaufrufe definiert:

Name	Nummer	Bedeutung
ERR_NO_ERROR	0	Der Systemaufruf konnte ohne Fehler ausgeführt werden.
ERR_NOT_ROOT	1	Dieser Systemaufruf kann (mit den gegebenen Parametern) nur von Root ausgeführt werden.
ERR_ACCESS_DENIED	2	Der Zugriff auf diesen Thread ist verboten.
ERR_RESOURCE_BUSY	3	Der Zugriff afu diesen Thread ist temporär nicht möglich, da er auf eine bestimmte SID eingeschränkt wurde oder anderweitig blockiert ist.
ERR_TIMED_OUT	4	Ein Timeout wurde erreicht.
ERR_INVALID_ARGUMENT	5	Ein allgemein ungültiger Parameter wurde übergeben.
ERR_INVALID_SID	6	Eine ungültige SID wurde als Parameter übergeben.
ERR_INVALID_ADDRESS	7	Eine ungültige Adresse wurde als Parameter angegeben.
ERR_NOT_ENOUGH_MEMORY	8	Es existiert kein weiterer Kernelspeicher, um eine bestimmte Operation durchzuführen.
ERR_PAGES_LOCKED	9	Ein Speicherbereich enthält bereits Speicherseiten, die zuerst freigegeben werden müssen, ehe die gewählte Operation darauf angewendet werden kann.
ERR_PAGING_DAEMON	10	Für eine bestimmte Operation wäre ein funktionierender Paging-Dämon erforderlich gewesen oder es wurde eine Operation ausgeführt, die nur dem Paging-Dämon erlaubt ist.
ERR_SYSCALL_RESTRICTED	11	Für diesen Systemaufruf existieren bestimmte plattformspezifische Einschränkungen, die überschritten worden sind.

Teil II

Schnittstellen der x86-Implementierung

Kapitel 6

Architekturabhängige Verfahrensweisen

6.1 Die Zielplattform

Die x86-Architektur ist eine Architektur mit einer langen Entwicklungsgeschichte. Sie fängt an mit der 8- und 16-bit Architektur der 8086er und der 80286er Serie, bis sie schließlich mit dem 80386 zu einer 32-bit Architektur erweitert wurde und heute mit der x86-64-Architektur von AMD letztlich auch als 64-bit Archiektur erhältlich ist.

Wenn in diesem Dokument von der x86-Architektur gesprochen wird, so werden damit nur alle 32-bit x86 Prozessoren ab der 80586-Serie bezeichnet. Die 80386 und 80486-er Serien werden bewusst nicht mehr unterstützt, um höhere Performance bei den heute noch gängigen x86-Architekturen zu erzielen und die Komplexität des Kernels so gering wie möglich zu halten (so ist z.B. keine FPU-Emulation erforderlich, da jede x86-CPU ab dem 80586 über eine eingebaute FPU verfügt). Die x86-64-Architektur von AMD, sowie die IA64-Architektur werden bei der x86-Implementierung des *hymk* nur in deren vollständigen 32-bit-Kompatibilitätsmodi unterstützt. Eine direkte 64-bit-Unterstützung würde eine Portierung des Kernels auf diese neuen Archietkuren erfordern, da sie sich von der bisher gängigen 32-bit Intel-Architektur (IA32) zu stark unterscheiden. Eine solche Portierung ist in Zukunft aber durchaus denkbar.

6.2 Der Startvorgang

Der Kernel wird auf der x86-Architektur durch den im Bereich der freien Software üblichen Bootloader „GRUB“ gestartet. Dieser Bootloader lädt neben dem Kernel noch weitere Module in den Arbeitsspeicher, die für den Betrieb des Gesamtsystems erforderlich sind.

6.3 Aufbau eines virtuellen Adressraums

Ein virtueller Adressraum hat in der x86-Architektur 4 GiB größe und ist in folgende Bereiche gegliedert:

Adressbereich	**Bedeutung**
`0x00000000 - 0x00000FFF`	„Zero-Page“ - Der Zugriff auf diese Seite ist gesperrt, um NULL-Pointer leichter aufzudecken
`0x00001000 - 0xBFFFEFFF`	Allgemeiner Benutzeradressraum
`0xBFFFF000 - 0xBFFFFFFF`	*Thread local storage* des aktuellen Threads (kein Caching; TLB wird bei jedem Kontext-Wechsel für diese Seite geleert)

Adressbereich	Bedeutung
0xC0000000 - 0xF7FFFFF	Gesperrter Kernel-Adressraum (Mapping der unteren 896 MiB des physikalischen Adressraums)
0xF8000000 - 0xF8000FFF	Haupt-Infopage
0xF8001000 - 0xFB000FFF	Prozesstabelle (Mapping der Prozessdeskriptoren; Im User-Mode nur lesbar, im Kernel-Mode R/W-Zugriff möglich)
0xFB001000 - 0xFE000FFF	Threadtabelle (Mapping der Prozessdeskriptoren; Im User-Mode nur lesbar, im Kernel-Mode R/W-Zugriff möglich)
0xFE001000 - 0xFFFDFFFF	Gesperrter Info-Page-Bereich (für spätere Versionen freigehalten)
0xFFFE0000 - 0xFFFFEFFF	Speichertransferpuffer (wird vom Kernel für Usermode-Speicherzugriffe verwendet - z.B. beim Copy-On-Write; kurz: UMCA)
0xFFFFF000 - 0xFFFFFFFF	„Last Page“ - Der Zugriff auf diese Seite ist gesperrt, um NULL-Pointer zu vermeiden

6.4 Caching und TLBs

Der Kernel verwendet die Caching-Fähigkeiten und die Translation-Lookaside-Buffer der x86-Architektur. Für Speicherseiten, die durch *io_alloc* in einen virtuellen Adressraum gemapt wurden, wird normalerweise der Cache deaktiviert - er kann aber auf Wunsch auch aktiviert werden. Normale Speicherseiten werden immer mit aktiviertem Cache gemapt.

Der *thread local storage* wird zwar mit aktiviertem Cache in einen virtuellen Adressraum eingeblendet, jedoch sorgt der Kernel mit der INVPLG-Instruktion der CPU dafür, dass der jeweilige TLB-Eintrag gelöscht wird, wenn ein Kontext-Wechsel zwischen Threads des gleichen Adressraums stattfindet. Gleiches gilt für die Info-Pages der Prozess- und Threadtabellen. Auch bei diesen wird automatisch eine INVPLG-Instruktion ausgeführt, sobald sich ein Mapping ändert.

Anonsten wird der TLB automatisch bei jeder *map*, *unmap* oder *move* Operation automatisch geleert. TLB und Cache werden ebenfalls mit jedem Wechsel eines virtuellen Adressraums geleert.

Die unteren 896 MiB des Kernel-Adressraums, sowie alle Info-Pages werden durch das sog. Global-Flag der 80686-Architekturen markiert. Dieses Flag bewirkt, dass die CPU bei einem Adressraumwechsel den TLB für diese Speicherseiten nicht leert, was ebenfalls die Performance erhöhen kann. Da der Kernel ja statisch in jedem Adressraum liegt, ist dies kein größeres Problem.

6.5 Systemaufrufe

Die Systemaufrufe des *hymk* werden durch Software-Interrupts mittels der INT-Instruktion der x86-Architektur realsiert. Während eines Systemaufrufs sie alle externen Interrupts gesperrt, da der Kernel derzeit nicth preemptiv implementiert ist. Nach einigen Systemaufrufen kann ein Kontext-Wechsel erfolgen, wodurch nicht sichergestellt sein kann, dass zwei aufeinanderfolgende Systemaufrufe - selbst bei gesperrten Interrupts - direkt nacheinander ausgeführt werden.

Die Systemaufrufe belegen die Interrupt-Vektoren *0xC0* bis *0xD5*. Eine Ausweitung des Bereichs ist in späteren Versionen möglich.

6.6 IRQs

Die 16 möglichen IRQs der x86-Architektur werden auf niedrigster Ebene im Kernel behandelt, aber von diesem an den Thread weitergegeben, der sich durch den Aufruf *recv_irqs* zur Behandlung eines IRQs gemeldet hat. Während ein Thread einen IRQ behandelt, wird die Behandlung des IRQs auf dem Interrupt-Controler der x86-Architektur (*Programmable interrupt controler* - „PIC“) gesperrt - es ist aber durchaus möglich, dass andere IRQs eintreten können. Somit dürfen IRQ-Behandlungsroutinen auch blockierende Operationen, wie *sync*, ausführen.

Der IRQ 0, der Interrupt der Systemuhr, wird vom Kernel mitverwendet, um die verbleibende CPU-Zeit eines Threads zu messen und ggf. (bei Ablauf dieser) einen Threadwechsel durchzuführen. Der IRQ 0 wird selbst bei Behandlung im Benutzer-Modus durch einen Thread nicht blockiert, wodurch durchaus während der Behandlung des IRQ 0 im Benutzermodus weitere Ereignisse des IRQs eintreten können, die von keinem weiteren Thread im Benutzer-Modues behandelt werden können. Daher kann z.B. durch die Behandlung des IRQ 0 im Benutzermodus nicht verlässlich als Zeitsignal zur Zeitmessung verwendet werden.

Die Software-Interrupts der IRQs leigen bei den Interrupt-Nummern 0xA0 bis 0xAF. Sie sind für den Zugriff aus dem Benutzermodus gesperrt. Versucht ein Thread sie auszuführen löst er eine Schutzverletzung aus - es sei denn, seine Software-Interrupts werden durch einen anderen Thread mittels *recv_softints* behandelt.

Wichtig bei der Behandlung eines IRQs ist, dass der selbe Thread anschließend den gleichen IRQ wieder mit *recv_irqs* überwacht oder ihn durch *recv_irqs(0xFFFFFFFF)* freigibt - ansonsten bleibt der IRQ bis dahin gesperrt.

6.7 Exceptions

Exceptions werden auf unterster Ebene durch den Kernel behandelt. Sie belegen die Software-Interrupts *0x00* bis *0x20*, auch wenn die Intel-Architektur nicht alle davon verwendet. Die Exceptions sind in der Regel ebenfalls für den Zugriff aus dem Benutzermodus gesperrt. Einzige Ausnahme bilden die Softwareinterrupt 3, 4 und 5 die u.a. auch durch die Instruktionen *INTO* und *INT3* aufgerufen werden können. Alle Exceptions können durch *recv_softints* in den Benutzermodus umgeleitet werden, sowohl wenn sie durch einen tatsächliche Ausnahmefehler ausgelöst wurden, als auch wenn sie durch einen Software-Interrupt mittels der *INT*-Instruktion, *INTO* oder *INT3* ausgelöst wurden.

6.8 Ungenutzte Softwareinterrupts

Der Aufruf der verbleibenden Interrupt-Vektoren ist prinzipiell zulässig. Der Kernel führt beim Aufruf eines unbenutzten Vektors keine weitere Operation durch. Es findet nur ein kurzer Wechsel zwischen Kernel- und Benutzer-Modus statt. Die ungenutzten Softwareinterrupts können ebenfalls durch *recv_softints* umgeleitet werden.

6.9 Kernel-Debugging

Der Kernel bietet selbst kaum externe Eingriffsmöglichkeiten zum Kernel-Debugging. Ist der Kernel jedoch im Debugging-Modus kompiliert worden, so gibt er eigenständig Fehlermeldungen aus.

Kapitel 7

Die x86-Systemaufrufe

7.1 Verwaltungsaufrufe

7.1.1 alloc_pages

Interrupt:

0xC0

Eingabeparameter:

EAX:	Startadresse des Mapping
EBX:	Anzahl der zu reservierenden Speicherseiten

Rückgabewerte:

EAX:	Fehlercode

Beschreibung:

Reserviert freie Seitenrahmen und fügt sie an der festgelegten Startadresse im aktuellen virtuellen Adressraum ein. Enthält ein Teil des Zielbereichs im aktuellen Adressraum bereits Seitenrahmen, so werden diese nicht überschrieben. Der Zielbereich darf weder ganz, noch teilweise im Kernel-Adressraum liegen. Er darf ebenfalls nicht im Bereich des *thread local storage* liegen. Der neue Speicherbereich wird als Lesbar, Beschreibbar und Ausführbar in den Adressraum eingeblendet.

Alle Bytes jeder neuegemappten Seite werden mit dem Wert „0“ aufgefüllt. Bestehende Mappings, die von *alloc_pages* nicht ersetzt werden können, werden jedoch dabei nicht überschrieben.

Beschränkungen:

Um einen zu langen Aufenthalt des Systems im Kernel-Mode zu verhindern, kann nur eine plattformspezifische Menge an Speicherseiten auf einmal alloziiert werden. Für größere Bereiche sind mehrere Allokationen in Folge erforderlich. Die Anzahl der maximal alloziierbaren Seiten kann aus der Hauptinfopage entnommen werden.

Fehlercodes:

`ERR_NOT_ENOUGH_MEMORY`	Es steht nicht genug Speicher zur Verfügung.
`ERR_SYSCALL_RESTRICTED`	Es können nicht mehr als eine gewisse Anzahl von Seiten (normalerweise 8 MiB) auf einmal alloziiert werden. Es fand keine Operation statt.
`ERR_INVALID_ADDRESS`	Der Zielbereich lag ganz oder teilweise im Kernel-Adressraum oder im *thread local storage*.

7.1.2 create_thread

Interrupt:

0xC1

Eingabeparameter:

EAX:	Startadresse des neuen Threads
EBX:	Anfangsadresse des Stacks des neuen Threads

Rückgabewerte:

EAX:	Fehlercode
EBX:	SID des neuen Threads

Beschreibung:

Erzeugt einen neuen Thread, der Anfangs noch durch *freeze_subject* eingefroren ist. Dieser neue Thread erbt von dem Thread, der ihn durch diesen Aufruf erstellt hat, dessen statische Priorität und dessen Schedulingklasse.

Fehlercodes:

`ERR_NOT_ENOUGH_MEMORY`	Es steht nicht genug Speicher zur Verfügung.

7.1.3 create_process

Interrupt:

0xC2

Eingabeparameter:

EAX:	Startadresse des neuen Threads
EBX:	Anfangsadresse des Stacks des neuen Threads

Rückgabewerte:

EAX:	Fehlercode
EBX:	SID des Controller-Threads des neuen Prozesses

Beschreibung:

Erzeugt einen neuen Prozess mit leerem virtuellem Adressraum und richtet für ihn einen neuen Thread ein, der durch *freeze_subject* eingefroren ist, aber vom Erzeugerprozess gesendete map-Operationen gestattet. Der neue Thread erbt dabei sie statische Priorität und die Schedulingklasse seines Erzeugers. Der neue Thread wird automatisch als *controller thread* des neuen Prozesses verwendet.

Fehlercodes:

`ERR_NOT_ENOUGH_MEMORY`	Es steht nicht genug Speicher zur Verfügung.

7.1.4 set_controller

Interrupt:

0xC3

Eingabeparameter:

EAX: SID des neuen Controller-Threads

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Legt den Controller-Thread des aktuellen Prozesses fest. Der Thread muss selbstverständlich ein Thread des selben Prozesses sein.

Fehlercodes:

`ERR_INVALID_SID`	Die übergebene SID ist ungültig oder beschreibt keinen Thread, der zu diesem Prozess gehört.

7.1.5 destroy_subject

Interrupt:

0xC4

Eingabeparameter:

EAX: SID des betroffenen Prozess- oder Thread-Subjekts

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Vernichtet ein Subjekt und gibt seine Datenstrukturen frei.

Ein Prozess-Subjekt wird dabei allerdings nicht unmittelbar aufgelöst, sondern durch setzen eines Flags blockiert, so dass dessen Threads nicht mehr ausgeführt werden können. Dies ist sinnvoll, um die Threads eines Prozesses bereits vollständig und irreversibel zu blockieren, um dann Schritt für Schritt dessen Threads zu beenden. Dieser Blockade-Zustand wird als „Zombie-Modus“ oder „Defunct-Modus“ bezeichnet.

Ein Prozess wird erst dann tatsächlich aufgelöst, wenn alle seine Threads beendet wurden. Der Kernel gibt bei dieser Auflösung nicht die Speicherseiten des Prozesses frei - hierfür ist der Paging-Dämon zuständig. Der Kernel kümmert sich nur um die Freigabe der Deskriptoren, Seitentabellen und Seitenverzeichnisse.

Ein Thread-Subjekt wird bei diesem Vorgang tatsächlich beendet und alle dessen Datenstrukturen, wie Deskriptor, Kernelstack und *thread local storage* werden dabei vom Kernel freigegeben. War der Thread der *controller thread* seines Prozesses, so setzt der Kernel den Zeiger auf den *controller thread* des Prozesses auf *invalid*. War der Thread der letzte Thread seines Prozesses, so wird der zugehörige Prozess anschließend auch aufgelöst (siehe oben).

Ein Thread kann sich durch den Aufruf von *destroy_subject* nicht selbst beenden. Ein Thread kann nur die Threads beenden, die zum selben Prozess gehören. Ist ein Thread Teil eines *root*-Prozesses, so ist er auch in der Lage, fremde Threads zu beenden. Ein Prozess kann sich ebenfalls nicht durch den Aufruf von *destroy_subject* selbst in den Zombie-Modus versetzen. Hierfür ist der Aufruf von *destroy_subject* durch einen Root-Prozess erforderlich. Unabhängig davon können sich auch *root*-Thrads und *root*-Prozesse nicht selbst beenden. Das Ziel hierbei ist, ein unkontrolliertes Hinterlassen von Systemressourcen durch eine kontrollierte Beendigung der Prozesse zu vermeiden.

Versuchte ein anderer Thread sich mit dem zu zerstörenden Thread zu synchronisieren, so wird dessen Synchronisierung abgebrochen. Dies betrifft jedoch nur Threads, die sich ausdrücklich mit einem anderen Thread synchronisieren. Bei Synchronisation mit einem Prozess wird selbst bei Beendigung von diesem keine Aktion getätigt.

Beschränkungen:

Bei Beendigung von Threads darf nur ein Thread des gleichen Prozesses einen anderen Thread beenden. *root*-Threads dürfen grundsätzlich alle Threads beenden. Ein Thread darf sich jedoch niemals selbst beenden.

Prozesse können ausschließlich nur durch *root*-Threads beendet werden. Ein Prozess kann sich dabei auch nicht selbst beenden.

Fehlercodes:

`ERR_ACCESS_DENIED`	Nur Threads des gleichen Prozesses und *root*-Threads dürfen diesen Systemaufruf tätigen.
`ERR_INVALID_SID`	Die verwendete SID ist ungültig.

7.2 Sicherheitsfunktionen

7.2.1 chg_root

Interrupt:

0xC5

Eingabeparameter:

EAX:	SID des Prozess-Subjekt, das seinen root-Modus ändern soll.	
EBX:	Subjekt	
	0	soll den Root-Modus verlassen.
	1	soll den Root-Modus betreten.

Rückgabewerte:

EAX:	Fehlercode

Beschreibung:

Wechselt einen Prozess in den Root-Modus oder entfernt ihn aus diesem. Beim Entfernen aus dem Root-Modus werden die I/O-Zugriffsrechte automatisch zurückgesetzt, d.h. der Zugriff auf I/O-Ports ist gesperrt.

Beschränkungen:

Diese Operation ist nur Root-Prozessen erlaubt.

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die verwendete Prozess-SID ist ungültig.

7.3 Scheduler

7.3.1 freeze_subject

Interrupt:

0xC6

Eingabeparameter:

EAX: SID des Threadsubjekts, das eingefroren werden soll (*null* oder *invaild* für aktuellen Thread)

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Friert einen Thread ein. Dabei kann auch der aktuelle Thread eingefroren werden.

Wichtig ist, dass ein mehrfaches Einfrieren eines Subjekts dazu führt, dass es auch wieder mehrfach mit *awake_subject* aufgeweckt werden muss.

Das Einfrieren von Prozessen ist derzeit nicht möglich. Ebenfalls können Threads, die sich im Kernel-Modus befinden (z.B. weil sie auf eingehende Synchronisierungen warten) nicht eingefroren werden.

Beschränkungen:

Diese Operation ist nur Root-Prozessen vorbehalten. Threads des gleichen Prozesses können diese Operationen ebenfalls untereinander anwenden. Der Zugriff auf den Kernel-Thread wird verweigert.

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die verwendete Ziel-SID ist ungültig.
`ERR_NOT_ENOUGH_MEMORY`	Das Subjekt wurde bereits so oft eingefroren, dass der zuständige Counter nicht mehr ausreicht.

7.3.2 awake_subject

Interrupt:

0xC7

Eingabeparameter:

EAX: SID des Threadsubjekt, das aufgeweckt werden soll.

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Weckt einen eingefrorenen Thread wieder auf. Wichtig ist, dass jedes *freeze_subject* auf ein Subjekt ein *awake_subject* benötigt. D.h. wurde ein Subjekt zweimal eingefroren, muss es auch zwei Mal wieder aufgeweckt werden, ehe der Kernel es wieder in die Run-Queue des Schedulers aufnimmt.

Beschränkungen:

Diese Operation ist nur Root-Prozessen vorbehalten. Threads des gleichen Prozesses können diese Operationen ebenfalls untereinander anwenden. Der Zugriff auf den Kernel-Thread wird verweigert. Das Aufwecken von Prozessen ist derzeit nicht möglich, kann aber ggf. in späteren Versionen eingeführt werden.

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die verwendete Ziel-SID ist ungültig.
`ERR_INVALID_ARGUMENT`	Das betroffene Subjekt ist bereits aktiv.

7.3.3 yield_thread

Interrupt:

0xC8

Eingabeparameter:

EAX: SID des Threads, der den Rest der effektiven Priorität empfangen soll

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Gibt die restliche effektive Priorität (also die CPU Zeit) des aktuellen Threads ab, so dass dieser die CPU solange verliert, bis alle anderen rechenbereiten Threads vom Scheduler einmal aufgerufen worden sind[1]. Die restliche effektive Priorität kann dabei auch an einen anderen Thread weitergegeben werden, so dass dessen Chance erhöht wird, sofort nach dem aktuellen Thread aktiviert zu werden.

Fehlercodes:

`ERR_INVALID_SID`	Die verwendete Ziel-SID ist ungültig.

[1] Wenn ein Thread, der mit *yield_thread* kurzzeitig die CPU abgegeben hat, von einem anderen Thread wieder ausreichend effektive Priorität über dessen *yield_thread*-Operation erhält, so kann es passieren, dass er erneut aktiviert wird, bevor alle anderen Threads vom Scheduler bearbeitet wurden.

7.3.4 set_priority

Interrupt:

0xC9

Eingabeparameter:

EAX:	SID des betroffenen Threads
EBX:	Neue Priorität (Zahl von 0 bis 40)
ECX:	Neue Schedulingklasse
	0 SCHED_REGULAR

Rückgabewerte:

EAX:	Fehlercode

Beschreibung:

Ändert die statische Priorität eines Threads, sowie dessen Scheduling-Klasse (derzeit existiert nur eine).

Beschränkungen:

Nur Root-Prozesse können die Priorität oder die Scheduling-Klasse anheben. Andere Prozesse können sie nur absenken. Der Zugriff auf den Kernel-Thread wird verweigert.

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die verwendete Ziel-SID ist ungültig.
`ERR_INVALID_ARGUMENT`	Die verwendete Priorität oder Scheduling-Klasse ist ungültig.

7.4 Gemeinsame Speichernutzung

7.4.1 allow

Interrupt:

0xCA

Eingabeparameter:

EAX: SID die für die Erlaubnis verwendet werden soll. Dies muss eine Thread-SID, eine Prozess-SID oder eine Prozessgruppen-SID sein. Wird *null* oder *invalid* verwendet, ist der Zugriff nur für den aktuellen Thread möglich - für alle anderen Threads ist er gesperrt. *kernel* hat keine Wirkung. Die SID muss nicht gültig sein.

EBX: SID des Threads, der die Allow-Operation ausführen soll (wichtig für Root-Threads). Soll die Operation im Namen des aktuellen Threads ausgeführt werden (Normalfall), so wird einfach *invalid* als SID übergeben.

ECX: Startadresse des für die Operation genehmigten Bereichs

EDX: Anzahl der Seiten des Bereichs

EDI: Erlaubte Operationen (als Flags, die parallel gesetzt werden können):

`ALLOW_MAP`	1	Es ist erlaubt Seiten in den festgelegten Adressrbereich des virtuellen Adressraums des betroffenen Threads hineinzumappen (die Operation *move* eingeschlossen).
`ALLOW_UNMAP`	2	Es ist erlaubt Seiten aus dem festgelegten Adressbereich des virtuellen Adressraums des betroffenen Threads zu entfernen oder die Zugriffsrechte darauf einzuschränken.
`ALLOW_REVERSE`	5	Es ist erlaubt, Seiten mittels *MAP_REVERSE* aus dem Zieladressraum in den Quelladressraum zu mappen. (Impliziert *ALLOW_MAP*)

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Führt auf einen Thread die *allow* Operation aus, so dass auf einen bestimmten Speicherbereich des virtuellen Adressraums diees Threads die Speicheroperationen *map*, *unmap* und *move* angewendet werden können. Normalerweise führt ein Thread die Operation nur für sich selbst aus und setzt dabei als Ziel-SID auch *invalid*. Die Freigabe des Kernel-Adressraums ist nicht zulässig.

Beschränkungen:

Die *allow* Operation kann normalerweise nur auf den eigenen Thread angewendet werden. Root-Prozesse haben das Recht die Operation auch auf andere Threads anzuwenden.

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_ARGUMENT`	Die verwendeten Operationsflags sind ungültig.
`ERR_INVALID_ADDRESS`	Die Zieladresse überschneidet sich mit dem Kernel-Adressraum.

7.4.2 map

Interrupt:

0xCB

Eingabeparameter:

EAX: SID des Threads auf den die Operation angewandt werden soll (*null* entspricht dem eigenen Thread)

EBX: Startadresse des Quellbereichs im eigenen Adressraum

ECX: Anzahl der Seiten des Bereichs

EDX: Flags:

`MAP_READ`	1	Die Seiten können ausgelesen werden.
`MAP_WRITE`	2	Die Seiten können beschrieben werden (auf x86 impliziert dies MAP_READ).
`MAP_EXECUTABLE`	4	Die Speicherseiten können ausgeführt werden (auf x86 nicht verfügbar).
`MAP_COPYONWRITE`	8	Die Speicherseiten sind für das Copy-On-Write Verfahren markiert, so dass effektiv kein gemeinsamer Speicherbereich, sondern eine Kopie des Speicherbereichs entsteht. Hierbei werden ebenfalls die Seiten des Quellbereich für Copy-On-Write markiert!
`MAP_PAGED`	16	Die Speicherseite ist durch den Paging-Dämon geschützt. Ein Unmappen der Seite ist nicht gestattet (das Flag darf nur der Paging-Dämon verwenden).
`MAP_REVERSE`	32	Der Mapping-Vorgang soll umgekehrt werden: Die Seiten des Zieladressraums sollen in den Quelladressraum gemappt werden.

EDI Offsetadresse im Zielbereich

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Richtet eine gemeinsame Speichernutzung aus dem gegebenen Speicherbereich des Adressraums des aktuellen Threads unter bestimmten Flags mit dem Adressraum eines Ziel-Threads ein. Das Mappen aus dem Kernel-Adressraum ist dabei nicht zulässig.

Ebenfalls dürfen bestehende Zugriffsbeschränkungen des eigenen Adressraums (z.B. Nur-Lesbarkeit eines Adressbereichs) nur dann im Zieladressraum durch die Flags aufgehoben werden, wenn der aufrufende Thread Teil eines Root-Prozesses ist (d.h. ist der zu mappende Bereich „nur lesbar“, so können nicht-Root-Prozesse ihn auch nur als „nur lesbar“ weitergeben). Das Erhöhen der Privilegien wird lediglich unterbunden und führt nicht zu einem Abbruch der Map-Operation. Root-Prozesse können dagegen fehlende Zugriffsrechte einer Speicherseite hinzufügen. Eigenschaften, wie ein abgeschaltetes Caching werden „vererbt“.

Ist eine Seite im Quellmapping für Copy-On-Write markiert, so wird die Copy-On-Write-Prozedur für diese Seite im Quellmapping ausgelöst, um einen undefinierten Zustand der Seite bei der gemeinsamen Nutzung zu vermeiden. Dieser Vorgang wird jedoch unterbunden, wenn das Flag *MAP_COPYONWRITE* gesetzt ist und das Zielmapping somit eine auf Copy-On-Write basierte Kopie des Quellmappings darstellen soll.

Enthält ein Teil des Zieladressraums bereits Seitenrahmen, so bleiben diese bestehen. D.h. bevor in einen Bereich gemapt werden kann, muss sichergestellt sein, dass er keine Speicherseiten mehr enthält (dies ist die Aufgabe einer sauberen Speicherverwaltung im Benutzermodus).

Das Flag *MAP_REVERSE* kehrt die Richtung des Vorgangs um. Das Zielmapping wird zum Quellmapping und vice versa. Alle genannten Einschränkungen und Sicherheitsrichtlinien gelten entsprechend umgekehrt.

Beschränkungen:

Die Operation erfordert, dass auf die Gegenseite zuvor eine *allow*-Operation ausgeführt wurde, so dass die Zieladresse und die maximale Zielgröße und die Zugriffsbeschränkungen bekannt sind. Der Quellbereich muss in den Zielbereich hineinpassen. Der Zugriff kann verweigert werden, wenn die Zugriffsbeschränkungen der vorausgegangene *allow*-Operation den Zugriff nicht gestattet hat.

Grundsätzlich kann nur eine bestimmte, plattformspezifische Anzahl von Speicherseiten auf einmal von der *map*-Operation verarbeitet werden. Der entsprechende Wert kann aus der Hauptinfoseite entnommen werden.

Fehlercodes:

ERR_ACCESS_DENIED	Die Operationsbeschränkungen wurden übertreten.
ERR_PAGING_DAEMON	Das Flag MAP_PAGED kann nur durch den Paging-Dämon verwendet werden.
ERR_PAGES_LOCKED	Die Copy-On-Write-Operation konnte für eine Seite nicht ausgeführt werden. Ursache könnte eine gesperrte Speicherseite oder mangelnder Arbeitsspeicher sein.
ERR_INVALID_SID	Die verwendete Ziel-SID ist ungültig.
ERR_SYSCALL_RESTRICTED	Es kann nur eine gewisse Anzahl von Seiten (normalerweise 8 MiB) auf einmal von *map* verarbeitet werden. Es fand keine Operation statt.
ERR_INVALID_ARGUMENT	Die verwendeten Operationsflags sind ungültig.
ERR_INVALID_ADDRESS	Die Quelladresse überschneidet sich mit dem Kernel-Adressraum.

7.4.3 unmap

Interrupt:

0xCC

Eingabeparameter:

EAX: SID des Threads auf den die Operation angewandt werden soll. Wird *null* oder *invalid* übergeben, wird als Ziel der aktuelle Thread verwendet.

EBX: Startadresse des Zielbereichs

ECX: Anzahl der Seiten des Bereichs

EDX: Flags:

`UNMAP_COMPLETE`	0	Die Seiten sollen vollständig aus dem Adressraum entfernt werden. Der physikalische Seitenrahmen wird freigegeben, wenn kein anderer Prozess ihn mehr benutzt. Hierbei wird das Schutz-Flag des Paging-Dämons verändert. Ist der Aufrufer nicht der Paging-Dämon, so wird eine Paging-Exception ausgelöst, sofern der Paging-Dämon die Seite als geschützt markiert.
`UNMAP_AVAILABLE`	1	Die Speicherseiten sollen als nicht-vorhanden markiert werden. Der physikalische Seitenrahmen wird freigegeben, wenn kein anderer Prozess ihn mehr benutzt. Das Schutz-Flag des Paging-Dämons bleibt dabei unberührt.
`UNMAP_WRITE`	2	Die Seiten dürfen nur noch ausgelesen werden.
`UNMAP_EXECUTE`	4	Die Speicherseiten dürfen nicht mehr ausgeführt werden. (auf x86 ineffektiv)

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Entfernt einen Speicherbereich aus einem Adressraum, der einem bestimmten Thread zugeordnet ist oder reduziert zumindest die Zugriffsrechte auf diesen. Der Zugriff auf den Kernel-Adressraum ist auf keinen Fall gestattet.

Beschränkungen:

Die Operation erfordert, dass auf die Gegenseite zuvor eine *allow*-Operation ausgeführt wurde, so dass die Zieladresse und die Zugriffsbeschränkungen bekannt sind. Die Zieladresse und die Größe müssen in dem Bereich liegen, der von der *allow*-Operation freigegeben wurde. Der Zugriff kann verweigert werden, wenn die Zugriffsbeschränkungen der vorausgegangene *allow*-Operation den Zugriff nicht gestattet hat.

Führt ein root-Prozess diese Operation aus, so kann er auch ohne vorangegangene *allow*-Operation *unmap* ausführen und ebenfalls den Zielbereich übertreten. Gleiches gilt ebenfalls, wenn ein Thread die Operation auf sich selbst ausführt.

Führt ein Thread die unmap-Operation auf sich selbst aus, kann dies ohne ein vorausgegangenes *allow* geschehen.

Es kann ferner nur eine gewisse, plattformspezifische Anzahl von Speicherseiten auf einmal von der *unmap*-Operation verarbeitet werden. Die Anzahl kann aus der Hauptinfopage entnommen werden.

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_PAGING_DAEMON`	Auf eine vom Paging-Dämon geschützte Seite konnte *unmap* nicht mit *UNMAP_COMPLETE* angewendet werden. Es wurde aber möglicherweise auf einige Seiten *unmap* ausgeführt. Dieser Fehler sollte nur bei sehr schwerwiegenden Systemfehlern auftreten und kann daher in der Regel vernachlässigt werden.
`ERR_INVALID_SID`	Die verwendete Ziel-SID ist ungültig.
`ERR_SYSCALL_RESTRICTED`	Es sollten mehr als die plattformspezifische Anzahl von Seiten (normalerweise 8 MiB) auf einmal mit *unmap* bearbeitet werden. Es fand keine Verarbeitung statt.
`ERR_INVALID_ARGUMENT`	Die verwendeten Operationsflags sind ungültig oder die Zieladresse überschneidet sich mit dem Kernel-Adressraum.

7.4.4 move

Interrupt:

0xCD

Eingabeparameter:

EAX:	SID des Threads auf den die Operation angewandt werden soll
EBX:	Startadresse des Quellbereichs im lokalen Adressraum
ECX:	Anzahl der Seiten des Bereichs
EDX:	Flags (siehe *map*)
EDI	Offsetadresse im Zielbereich

Rückgabewerte:

EAX:	Fehlercode

Beschreibung:

Verschiebt einen Adressraum des aktuellen Threads in einen Adressraum eines bestimmten Ziel-Threads. Das Verschieben ist eine Kombinierte *map* Operation auf den Zieladressraum und eine im Anschluß stattfindende *unmap* Operation auf den lokalen Adressraum. Das Verschieben von Teilen des Kernel-Adressraums ist nicht zulässig. Enthält ein Teil des Zieladressraums bereits Seitenrahmen, so werden diese freigegeben und der Bereich durch das Quellmapping ersetzt.

Beschränkungen:

Die Operation erfordert, dass auf die Gegenseite zuvor eine *allow*-Operation ausgeführt wurde, so dass die Zieladresse und die Zugriffsbeschränkungen bekannt sind. Der Quellbereich muss in den Zielbereich passen, der von der *allow*-Operation freigegeben wurde. Der Zugriff kann verweigert werden, wenn die Zugriffsbeschränkungen der vorausgegangene *allow*-Operation den Zugriff nicht gestattet hat.

Für die Anzahl der Seiten, die von *move* maximal auf einmal verarbeitet werden können, gilt das gleiche, wie für *map* und *unmap*.

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die verwendete Ziel-SID ist ungültig.
`ERR_SYSCALL_RESTRICTED`	Es sollten mehr als die plattformspezifische Anzahl von Seiten (normalerweise 8 MiB) auf einmal mit *move* bearbeitet werden. Es fand keine Verarbeitung statt.
`ERR_INVALID_ARGUMENT`	Die verwendeten Operationsflags sind ungültig oder die Zieladresse überschneidet sich mit dem Kernel-Adressraum.

7.5 Synchronisation

7.5.1 sync

Interrupt:

0xCE

Eingabeparameter:

EAX:	SID der Gegenseite
EBX:	Länge des Timeouts (ungefähr) in Millisekunden (0 = kein Warten, 0xFFFFFFFF = unendlich)
ECX:	Anzahl der Resync-Vorgänge (0 = kein Resync)

Rückgabewerte:

EAX:	Fehlercode
EBX:	SID des Threads, der sich mit dem Aufrufer synchronisiert hat

Beschreibung:

Synchronisiert zwei Threads innerhalb eines bestimmten (oder unendlichen) Timeouts miteinander - d.h. der Aufrufer wartet, bis ein gewählter anderer Thread (definiert durch den Parameter *SID*) ebenfalls *sync* mit seiner SID aufruft. Wird das Timeout 0 gesetzt, wird nicht auf die Gegenseite gewartet, sondern die Operation sofort wieder abgebrochen.

Das Warten geschieht weitgehend passiv, so dass durch den Wartevorgang keine oder relativ wenig CPU-Zeit verbraucht wird. Der Kernel kann zur Umsetzung interne Warteschleifen einsetzen. Gewartet wird solange, bis die Gegenseite die Synchronisierung gestattet - d.h. auch wenn die Gegenseite sich derzeit auf eingehende Synchronisationen wartet, aber die Synchronisation mit dem Aufrufer verbietet, wartet der Aufrufer, bis sich dieser Zustand der Gegenseite ändert oder das Timeout abläuft.

Die Gegenseite, auf die ein Thread wartet, kann

- ein bestimmter Thread (definiert durch dessen Thread-SID) sein
- ein beliebiger Thread eines bestimmten Prozesses (definiert durch die Prozess-SID) sein
- ein beliebiger Thread aller Prozesse sein, die sich im *root*-Modus befinden (definiert durch die *root*-SID)
- ein beliebiger Thread (definiert durch die *everybody*-SID) sein

Wichtig ist hierbei, dass bei einer Synchronisierung grundsätzlich einer der beiden Seiten eine konkrete Thread-SID angeben muss, da sonst das Ziel nicht genau spezifiziert ist.
Um unnötige Wechsel zwischen Kernel- und Benutzermodus zu vermeiden, kann ein Thread eine automatische Wiederholung des selben Synchronisationsvorgangs bei erfolgreicher Synchronisation anordnen. In einem Client-Server-System ist es z.B. oft der Fall, dass der Client sich zuerst mit dem Worker-Thread des Servers synchronisiert, um diesen zu signalisieren, dass ein neuer Auftrag ansteht und anschließend auf eingehende Synchronisation des Worker-Threads wartet, damit dieser ihm die Fertigstellung des Auftrags signalisiert. Durch die automatische Resynchronisierung kann hier ein überflüssiger Wechsel zwischen Kernel- und Benutzeradressraum vermieden werden.

Als Rückgabewert wird dem Aufrufer die Thread-SID der Gegenseite übermittelt, die sich tatsächlich synchronisiert hat.

Beschränkungen:

Die Operation erfordert, dass die Gegenseite den Zugriff von einer SID erlaubt, die dem jeweiligen aufrufenden Thread zugänglich ist (Thread-SID, Prozess-SID, *root*/*everybody*).

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die verwendete SID ist ungültig.
`ERR_INVALID_ARGUMENT`	Die verwendeten Operationsflags sind ungültig.
`ERR_TIMED_OUT`	Das Timeout wurde erreicht.
`ERR_RESOURCE_BUSY`	Die Warteschlange für den Ziel-Thread ist voll. Ein passives Warten ist derzeit nicht möglich. Es wird aktives Warten im Benutzer-Modus empfohlen.

7.6 Eingabe/Ausgabe

7.6.1 io_allow

Interrupt:

0xCF

Eingabeparameter:

EAX: SID des Zielprozesses

EBX: Flags:

`IO_ALLOW_IRQ`	1	Der Prozess hat nun das Recht IRQs zu behandeln, auch wenn er kein Root-Prozess ist.
`IO_ALLOW_PORTS`	2	Der Prozess hat nun das Recht auf I/O-Ports zuzugreifen, auch wenn er kein Root-Prozess ist.

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Erlaubt einem Nicht-Root-Prozess bestimmte I/O-Systemaufrufe und I/O-Instruktionen auszuführen, die sonst nur Root-Prozessen vorbehalten sind. Dadurch lassen sich Treiber realisieren, die nicht im Root-Modus laufen.

Beschränkungen:

Die Operation kann nur von Root-Prozessen ausgeführt werden.

Fehlercodes:

`ERR_NOT_ROOT`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die verwendete Prozess-SID ist ungültig.
`ERR_INVALID_ARGUMENT`	Die verwendeten Operationsflags sind ungültig.

7.6.2 io_alloc

Interrupt:

0xD0

Eingabeparameter:

EAX: Zu reservierender Quellspeicherbereich

EBX: Zielspeicherbereich des Mappings

ECX: Anzahl der zu mappenden Seiten

EDX: Flags:

`IOMAP_READ`	1	Der gemappte Hardwarespeicherbereich darf ausgelesen werden.
`IOMAP_WRITE`	2	Der gemappte Hardwarespeicherbereich darf beschrieben werden (impliziert auf x86 IOMAP_READ).
`IOMAP_EXECUTE`	4	Der gemappte Hardwarespeicherbereich darf ausgeführt werden (auf x86 ineffektiv).
`IOMAP_WITH_CACHE`	8	Aktiviert den Cache beim Mappen (sollte bei den meisten Hardwarespeicherbreichen nicht verwendet werden!)

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Mappt einen Hardwarespeicherbereich in den Adressraum des aktuellen Prozesses. Einige Speicherbereiche, z.B. Kernel-Code und Daten, sowie die PBT, FMT und der Page-Buffer können dabei nicht gemappt werden. Als Ziel kann nicht der Kerneladressraum verwendet werden.

Beschränkungen:

Die Operation kann nur von Root-Prozessen ausgeführt werden. Außerdem kann nur eine bestimmte plattformspezifische Anzahl von Seiten auf einmal von *io_alloc* verarbeitet werden. Die Anzahl kann aus der Hauptinfopage in Erfahrung gebracht werden.

Bestehende Seitenmappings werden nicht aufgelöst.

Fehlercodes:

`ERR_NOT_ROOT`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_ADDRESS`	Der verwendete Ziel- oder Quellbereich ist ungültig.
`ERR_SYSCALL_RESTRICTED`	Es sollten mehr als die plattformspezifische Anzahl von Seiten (normalerweise 8 MiB) auf einmal mit *io_alloc* bearbeitet werden. Es fand keine Verarbeitung statt.
`ERR_INVALID_ARGUMENT`	Die verwendeten Operationsflags sind ungültig.

7.6.3 recv_irq

Interrupt:

0xD1

Eingabeparameter:

EAX: Zu überwachender IRQ

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Lässt den aktuellen Thread auf einen eintreffenden IRQ warten. Wird während einer laufenden Behandlung des zu überwachenden IRQs dieser Aufruf auf denselben oder einen anderen IRQ gestartet, so wird die laufende Behandlung damit beendet und erneut auf den genannten IRQ gewartet, sofern nicht ein anderer Thread auf den IRQ wartet. Der Kernel übernimmt dabei auch die Informierung des PICs. Wird als zu überwachender IRQ während einer Behandlung die Nummer 0xFFFFFFFF angegeben, so wird die IRQ-Behandlung beendet, ohne auf neue IRQs zu warten.

Beschränkungen:

Die Operation kann nur von Root-Prozessen ausgeführt werden, oder von Prozessen denen das Recht der IRQ-Behandlung über *io_allow* zugestanden wurde.

Fehlercodes:

`ERR_ACCESS_DENIED`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_ARGUMENT`	Die verwendete IRQ-Nummer ist ungültig.
`ERR_RESOURCE_BUSY`	Der IRQ wird bereits von einem anderen Thread behandelt.

7.7 Thread-Fernsteuerung

7.7.1 recv_softints

Interrupt:

0xD2

Eingabeparameter:

EAX: Zu überwachender Thread

EBX: Timeout, (ungefähr) in Millisekunden (0 = kein Timeout, 0xFFFFFFFF unendliches Timeout)

ECX: Flags

RECV_AWAKE_OTHER	1	Der zu beobachtende Thread soll beim Start der Beobachtung mit *awake_subject* aufgeweckt werden.
RECV_TRACE_SYSCALL	2	Tritt ein Systemaufruf ein, wird dessen Behandlung nicht unterdrückt, sondern nur weitergemeldet. Anschließend wird der überwachte Thread angehalten. Andere Arten von Softwareinterrupts (insb. Exceptions) werden auch weiterhin unterdrückt.

Rückgabewerte:

EAX: Fehlercode

EBX: Eingetretener Software-Interrupt

Beschreibung:

Überwacht auftretende Software-Interrupts eines Ziel-Threads. Tritt ein Software-Interrupt ein, wird dies an den Überwacher weitergemeldet. Der Software-Interrupt wird dabei nicht vom Kernel behandelt, sondern der überwachende Thread kann die Behandlung durchführen. Der überwachte Thread wird zudem mit *freeze_subject* angehalten.

Wird das Flag *RECV_TRACE_SYSCALL* gesetzt, so werden vom Kernel alle Software-Interrupts behandelt, die Systemaufrufe ausführen. Unabhängig davon wird der Software-Interrupt dennoch weitergemeldet und wird nach dessen Ausführung der überwachte Thread angehalten.

Für die Operation kann ein Timeout gesetzt werden.

Beschränkungen:

Die Operation kann nur von Root-Prozessen ausgeführt werden. Der Zugriff auf den Kernel-Thread wird verweigert.

Fehlercodes:

`ERR_NOT_ROOT`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die SID des zu überwachenden Threads ist ungültig.
`ERR_TIMED_OUT`	Das Timeout endete.

7.7.2 read_regs

Interrupt:

0xD3

Eingabeparameter:

EAX:	SID des Threads, dessen Register gelesen werden sollen		
EBX:	Zu lesender Register-Typus		
	`REGS_X86_GENERIC`	0	(1) EAX
			(2) EBX
			(3) ECX
			(4) EDX
	`REGS_X86_INDEX`	1	(1) ESI
			(2) EDI
			(3) EBP
	`REGS_X86_POINTERS`	2	(1) ESP
			(2) EIP
	`REGS_X86_EFLAGS`	3	(1) EFLAGS

Rückgabewerte:

EAX:	Fehlercode
EBX:	(1)
ECX:	(2)
EDX:	(3)
ESI:	(4)

Beschreibung:

Liest einige Register eines Threads aus.

Beschränkungen:

Die Operation kann nur von Root-Prozessen ausgeführt werden. Der Zugriff auf den Kernel-Thread wird verweigert.

Fehlercodes:

`ERR_NOT_ROOT`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die SID des zu kontrollierenden Threads ist ungültig.
`ERR_INVALID_ARGUMENT`	Der gewählte Registertyp ist ungültig.

7.7.3 write_regs

Interrupt:

0xD4

Eingabeparameter:

EAX: SID des Threads, in dessen Register geschrieben werden sollen

EBX: Zu schreibender Register-Typus

`REGS_X86_GENERIC`	0	(1) EAX (2) EBX (3) ECX (4) EDX
`REGS_X86_INDEX`	1	(1) ESI (2) EDI (3) EBP
`REGS_X86_POINTERS`	2	(1) ESP (2) EIP
`REGS_X86_EFLAGS`	3	(1) EFLAGS Nur Änderung des ersten Bytes möglich

ECX: (1)

EDX: (2)

ESI: (3)

EDI: (4)

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Ändert die Register eines Threads.

Beschränkungen:

Die Operation kann nur von Root-Prozessen ausgeführt werden. Der Zugriff auf den Kernel-Thread wird verweigert.

Fehlercodes:

`ERR_NOT_ROOT`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die SID des zu kontrollierenden Threads ist ungültig.
`ERR_INVALID_ARGUMENT`	Der gewählte Registertyp ist ungültig.

7.8 Paging-Funktionen

7.8.1 set_paged

Interrupt:

0xD5

Eingabeparameter:

Keiner.

Rückgabewerte:

EAX: Fehlercode

Beschreibung:

Legt den aktuellen Thread als Paging-Dämon-Thread fest.

Beschränkungen:

Die Operation kann nur von Root-Prozessen ausgeführt werden.

Fehlercodes:

`ERR_NOT_ROOT`	Die Operationsbeschränkungen wurden übertreten.
`ERR_PAGING_DAEMON`	Es ist bereits ein Thread als Paging-Dämon eingetragen.

7.8.2 test_page

Interrupt:

0xD6

Eingabeparameter:

EAX	Adresse der Speicherseite
EBX	Prozess SID des Zieladressraums (*invalid* oder *null* für den aktuellen Adressraum)

Rückgabewerte:

EAX:	Fehlercode		
EBX:	Zugriffsflags der Page:		
	PGA_READ	1	Lesen ist erlaubt.
	PGA_WRITE	2	Schreiben ist erlaubt.
	PGA_EXECUTE	4	Ausführen ist erlaubt.

Beschreibung:

Gibt die Zugriffsrechte auf eine Speicherseite in einem Zieladressraum zurück.

Beschränkungen:

Die Operation kann nur von Root-Prozessen auf fremde Adressräume ausgeführt werden. Die Prüfung des eigenen Adressraums ist unbeschränkt.

Fehlercodes:

`ERR_NOT_ROOT`	Die Operationsbeschränkungen wurden übertreten.
`ERR_INVALID_SID`	Die SID des Zieladressraums ist ungültig.
`ERR_INVALID_ADDRESS`	Die angegebene Adresse liegt im Kernel-Adressraum.

Kapitel 8

Informationsseiten

8.1 Die Info-Pages

8.1.1 Die Hauptinfopage

Die Hauptinfopage ist bei der x86-Architektur an Adresse `0xF8000000` und ist wie jede andere x86-Seite 4096 Byte groß. Ihre Einträge sind 32-bit breit. Der Zugriff ist vom Kernel aus im Schreib-Lese-Modus und vom Benutzer-Modus aus im Lese-Modus möglich. Sie hat folgenden Aufbau:

- Die SID des aktuellen Prozesses
- Die SID des aktuellen Threads
- Einen Zeiger auf den Prozesstabelleneintrag des aktuellen Prozesses
- Einen Zeiger auf den Threadtabelleneintrag des aktuellen Threads
- Die Versionsnummer des Kernels
- SID des Paging-Dämons
- Der aktuelle RTC-Zähler (als 64-bit Wert verteilt über zwei 32-bit Einträge)
- Der Idendifikationscode für die CPU (z.B. 0x80386)
- Die Größe einer Speicherseite auf dieser Plattform
- Anzahl von Seiten, die maximal in einem Systemaufruf (z.B. alloc_pages, io_alloc, map)
- CPU-Daten:
  - CPU-Name (12-Byte)
  - CPU-Typ
  - CPU-Familie
  - CPU-Modell
  - Stepping-ID
  - Größe des physikalischen Adressraums

8.1.2 Die Prozesstabelle

Die Prozesstabelle liegt bei der x86-Architektur im Adressbereich `0xF8001000 - 0xFA000FFF`. Sie besteht aus Einträgen zu je zwei Pages, wobei eine Page den Prozessdeskriptor, und die zweite dessen Speicherverwaltungstabelle enthält. Da maximal 4096 Prozesse zulässig sind, hätte sie somit eine Größe von 32 MiB. Die Einträge für nicht aktive Prozesse werden mit dem gleichen leeren Speicherseitenrahmen abgedeckt. Die restlichen Einträge bestehen aus beliebig allokierten Seitenrahmen, die in diese Tabelle vom Kernel eingeblendet wurden. Sie ist vom Kernel aus im Schreib-Lese-Modus erreichbar, vom Benutzer-Modus im Nur-Lese-Modus.

Prozessdeskriptoren

Die Prozessdeskriptoren haben Einträge zu je 32-bit breite und folgenden Aufbau:

- Flag, das festlegt, ob der Deskriptor in gebraucht ist (1) oder nicht (0)
- Flag, das festlegt, ob der Prozess entgültig gestoppt wurde (1) oder nicht (0)
- SID des Prozesses
- SID des *controller threads*
- Zeiger auf den Deskriptor des *controller threads*
- Anzahl der verwendeten Speicherseiten
- I/O-Zugriffsrechte des Prozesses (IRQs, I/O-Ports)
- Seitenrahmen der obersten Seitentabelle (*page directory*) des Prozesses
- Flag, das festlegt, ob der Prozess ein Paging-Dämon ist (1) oder nicht (0)
- Zähler, der die Anzahl der Threads eines Prozesses angibt
- Zeiger auf den ersten Thread aus der Liste der Threads des Prozesses
- Eindeutige Prozess-ID

Die Speicherverwaltungstabelle

Die Speicherverwaltungstabelle des Kernels enthält 1024 Einträge, wobei jeder Eintrag für einen Eintrag im Page-Directory des Prozesses steht und somit auch für einen entsprechenden 4 MiB-Speicherbereich im virtuellen Adressraum. Im Unterschied zur Page-Directory gibt die Speicherverwaltungstabelle nicht die Adressen der jeweiligen Seitentabellen an, sondern wieviele Seitendeskriptoren in einer Seitentabelle derzeit aktiv sind und wieviele nicht. Dadurch ermittelt der Kernel u.a. wann eine Seitentabelle freigegeben werden darf udn wann nicht. Der Kernel-Adressraum ist hierin nicht abgebildet.

8.1.3 Die Thread-Tabelle

Die Threadtabelle liegt bei der x86-Architektur im Adressbereich `0xFB001000 - 0xFD000FFF`. Sie besteht aus Einträgen zu je zwei Pages, wobei eine Page den Threaddeskriptor und die andere dessen *thread local storage* enthält. Da maximal 4096 Threads zulässig sind, hat sie somit eine Größe von 32 MiB. Die Einträge für nicht aktive Threads werden mit dem gleichen leeren Speicherseitenrahmen abgedeckt. Die restlichen Einträge bestehen aus beliebig allokierten Seitenrahmen, die in diese Tabelle vom Kernel eingeblendet wurden. Sie ist vom Kernel aus im Schreib-Lese-Modus erreichbar, vom Benutzer-Modus im Nur-Lese-Modus.

Implementationsunabhängige Daten

- Flag, das festlegt, ob der jeweilige Deskriptor in gebrauch ist (1) oder nicht (0)
- SID des Threads
- SID des Prozesses
- SID-Berechtigung (bzw. Ziel-SID) für *sync*
- SID-Berechtigung für Speicheroperationen
- Zieladresse für Speicheroperationen
- Größe für Speicheroperationen
- Zulässige Speicheroperationen
- Erster Seitenrahmen des Kernel-Stacks des Threads (eine Adresse relativ zum Begin des Kernel-Adressraums)
- Zeiger auf den Deskriptor des Prozesses
- Weitere Flags:

`THRSTAT_BUSY`	1	Der Thread ist aus irgendeinem Grund blockiert und nicht rechenbereit (Grund als Flag sichtbar)
`THRSTAT_KERNEL_MODE`	2	Der Thread befindet sich im Kernel-Modus.
`THRSTAT_IRQ`	4	Der Thread wartet auf IRQs
`THRSTAT_IRQ_HANDLING`	8	Der Thread behandelt einen IRQ
`THRSTAT_SYNC`	16	Der Thread synchronisiert sich mit einem anderen Thread.
`THRSTAT_TIMEOUT`	32	Der Thread hat ein Timeout gesetzt, das der Scheduler überwachen soll.
`THRSTAT_RECV_SOFTINT`	64	Der Thread wartet auf eingehende Softwareinterrupts eines anderen Threads.
`THRSTAT_FREEZED`	128	Der Thread ist eingefroren
`THRSTAT_WAIT_HYPAGED`	256	Der Thread wartet auf den HyPageD

THRSTAT_PROC_DEFUNC	512	Der Dachprozess wurde mit *destroy_subject* beendet.
THRSTAT_TRACE_ONLY	1024	Sofern eine Überwachung der Software-Interrupts aktiviert ist, wird die Ausführung dieser nicht vom Überwacher unterbunden.
THRSTAT_OTHER_FREEZE		Der Thread wurde wegen THRSTAT_IRQ, THRSTAT_SYNC, THRSTAT_RECV_SOFTINT, THRSTAT_WAIT_HYPAGED oder THRSTAT_PROC_DEFUNC eingefroren. Dieses Makro ist kein Flag im eigentliches Sinne, sondern fast die o.g. Flags zusammen.

- Nummer des IRQs, falls der Thread auf IRQs wartet `0xFFFFFFFF` wenn nicht
- Zähler, der festlegt, wie oft eine *freeze_subject*-Operation auf den Thread ausgeführt wurde
- Zeiger auf vorheriges Element der Run-Queue des Scheduler, falls der Thread rechenbereit und Teil der Runqueue ist
- Zeiger auf nächstes Element der Run-Queue des Schedulers, falls der Thread rechenbereit und Teil der Runqueue ist
- SID des Threads, der Softwareinterrupts dieses Threads abgreift
- Effektive Priorität des Threads
- Statische Priorität des Threads
- Scheduling-Klasse des Threads
- Zeitpunkt an dem eine Timeout-Behaftete Operation (z.B. *sync* oder *recv_softints*) abgebrochen werden soll (64-bit Wert, der zwei Deskriptoreinträge verbraucht)
- Plattformunabhängige Nummer der letzten Exception (kann auch Page Fault sein)
- Nummer der letzten Exception (Wert plattformspezifisch)
- Ausführungsadresse bei letzter Exception (Wert plattformspezifisch)
- Fehlercode der letzten Exception
- Seitendeskriptor der Seite an der der letzte Pagefault auftrat (Wert plattformspezifisch)
- Zeiger auf den vorherigen Thread in der Liste der Threads des Prozesses
- Zeiger auf den nächsten Thread in der Liste der Threads des Prozesses
- Letzter ausgeführter Software-Interrupt (falls Umleitung durch *recv_softints*)
- Nummer des Threads, von dem der aktuelle Thread Software-Interrupts abgreift
- Eindeutige Thread-ID

Implementierungsspezifische Statusinformationen

- Zeiger auf vorherigen Eintrag einer Sync-Warteschlange, in der der Thread Mitglied ist
- Zeiger auf nächsten Eintrag einer Sync-Warteschlange, in der der Thread Mitglied ist
- Zeiger auf erstes Element der eigenen Sync-Warteschlange
- Zeiger auf vorherigen Eintrag in der Timeout-Warteschlange, sofern der Thread ein Timeout gesetzt hat
- Zeiger auf nächsten Eintrag in der Timeout-Warteschlange, sofern der Thread ein Timeout gesetzt hat

Plattformspezifische Statusinformationen

- Zu letzt verwendeter Wert des Stack-Pointers des Kernel-Stacks (eine Adresse relativ zum Begin des Kernel-Adressraums). Diese Information ist im 100. Doppelwort des Deskriptors abgespeichert.
- Die phyiskalische Adresse der Speicherseite des Thread Local Storage (101. Doppelwort)
- Die letzten 1024 Byte des Deskriptors sind für den FPU-Stack reserviert

Der *thread local storage*

Der *thread local storage* ist eine normale Speicherseite von 4096 Byte größe. Sie ist ebenfalls im Benutzeradressraum im Schreib-Lese-Modus eingeblendet. Die Inhalte werden vom jeweiligen Thread bzw. dessen API-Bibliothek bestimmt.

Teil III

Implementierung auf der x86-Architektur

Kapitel 9

Grundaspekte der Implementierung

9.1 Verwendete Schutzkonzepte

Die x86-Implementierung des *hymk* arbeitet vollständig im sog. „Protected Mode“ der x86-Architektur. In diesem Modus bietet die CPU eine Reihe von Schutzmechanismen, einen Paging-Mechanismus, sowie den Zugriff auf den vollen 32-bit Adressraum an. Da die CPU normalerweise in einem Kompatibilitätsmodus für die 8086-Architektur startet (der sog. „Real Mode“), muss beim Start des Systems der Modus gewechselt werden, was wiederum Aufgabe des verwendeten Bootloaders ist. Der Kernel übernimmt nach dem Start des Systems die volle Kontrolle über die verschiedenen, von der CPU im Protected Mode verwendeten Verwaltungstabellen (GDT, IDT etc.).

Die Details der x86-Architektur können in diesem Dokument nicht näher erwähnt werden. Nähere Informationen zur x86-Architektur können u.a. aus den Architektur-Handbüchern der Firma Intel entnommen werden. In diesem Kapitel wird tabelarisch erläutert, wie die verschiedenen Schutzmechanismen auf der x86-Architektur umgesetzt wurden.

Schutzmechanismus	**Implementierung**
Trennung der virtuellen Adressräume	Wechsel der Page-Directories bei einem Kontext-Wechsel durch Änderung des CR3-Registers im Kernel-Modus (das CR3-Register kann NUR im Kernel-Modus geändert werden).
Trennung von Kernel- und Benutzer-Modus	Über das *current privilege level* (CPL) der CPU, das die Ausführung bestimmter Instruktionen (etc.) einschränken kann. Der Kernel-Modus verwendet „Ring 0“ - der Benutzermodus „Ring 3“ (in letzterem sind Kernel-Speicherseiten und bestimmte Instruktionen der CPU gesperrt). Ein Wechsel in den Kernel-Modus findet immer bei IRQs und Exceptions statt. Die eigtl. IRQ-Behandlung durch den jeweiligen Gerätetreiber wird aber dann wieder im Benutzermodus, also Ring 3 durchgeführt. Ansonsten ist ein Wechsel nur durch einen Systemaufruf möglich.

Schutzmechanismus	Implementierung
Schutz der Systemaufrufe	Systemaufrufe werden über Softwareinterrupts realisiert. Die dafür zuständige Tabelle, die sog. IDT, kann nur vom Kernel verändert werden, so dass dieser festlegen kann, welcher Softwareinterrupt zulässig ist und zu welchem Systemaufruf ein Aufruf von diesem führt.
Schutz des Kernel-Adressraums	Der Kernel-Adressraum in einem virtuellen Adressraum ist durch setzen des „Supervisor“-Flags in den jeweiligen Pagetable/Pagedirectory-Einträgen möglich.
Schutz der Info-Pages	Die Info-Pages sind im Benutzermodus nur lesbar, da das Schreibzugriffsflag der jeweiligen Seiten deaktiviet ist. Im Kernel-Modus sind sie lesbar und beschreibbar.
Schutz der I/O-Ports	Die I/O-Ports werden über das *I/O privilege level* (IOPL) der CPU geregelt. Wird ein Thread ausgeführt, der auf die Ports zugreifen darf, so wird dessen IOPL so gesetzt, dass er Zugriff auf die Ports erhält. Die I/O-Permission-Bitmap, die zur Sperrung einzelner Ports gedacht ist, wird nicht verwendet.
Aufdeckung von Null-Pointern	Null-Pointerzugriffe werden durch Sperrung der ersten und letzten Speicherseite der Programme leichter aufgedeckt.
Schutz der regulären Pageframes und des Kernels	Der Adressraum des Kernels, sowie die regulären, zur Speicherung gedachten Seitenrahmen sind vom *io_alloc*-Mechanismus ausgesperrt.

9.2 Bootvorgang

Der *hymk* wird auf der x86-Plattform durch den Bootloader GRUB gestartet. Da dieser bereits Fähigkeiten zum Wechsel in den Protected Mode besitzt, enthält der *hymk* selbst keinen Code mehr um eigenständig in den Protected Mode zu gelangen. Der GRUB-Bootloader lädt den Kernel in einen Speicherbereich oberhalb des ersten MiBs des Arbeitsspeichers. Außerdem lädt GRUB weitere essentielle Systemserver im Anschluß daran (siehe GRUB Multiboot-Spezifikationen). Darunter ist auch das Image des Init-Prozesses. Anschließend startet er den Kernel durch einen Sprung an die erste Ausführungsadresse. Im Kernel-Code liegt an dieser Adresse der Einsprungspunkt *start* (siehe *start.s*). Das Programm an dieser reinitialisiert die GDT und IDT, wählt die Selektoren für die Kernel-Speichersegmente, setzt einen initialen Kernel-Stack und wechselt dann in die Routine *main* (siehe *init.c*). Die Routine *main* kümmert sich um die weitere Initialisierung des Systems. Die genauen Details dieser Initialisierung können der Dokumentation innerhalb des Quellcodes und den Dokumentationen der jeweiligen Systemteile (Speicherverwaltung etc.) entnommen werden.

Nachdem der Kernel sich initialisiert hat, lädt er das Image von Init in einen neuen Prozess und startet diesen anschließend. Init hat dann die Aufgabe weitere Systemdienste zu laden.

9.3 Verwendete Code-Konventionen

Vor dem weiteren Einstieg in die verwendeten Verfahren sollten noch einige Konventionen innerhalb des Kernel-Quellcodes erwähnt werden.

9.3.1 Standard-Konformität

Der Kernel-Quellcode ist nicht auf ISO-Konformität hin entwickelt worden, sondern hin zu den Erweiterungen des ISO-C99-Standards durch den GNU-C-Compiler (gcc/gnu99). Es werden u.a. Inline-Assembler und die vom gcc angebotenen Funktionsartigen Makros benutzt. Daher dürfte es nur schwer möglich sein den Kernel mit einem anderen Compiler, als dem gcc zu kompilieren.

Der im Kernel-Code verwendete Assemblercode wurde für den GAS-Assemblierer ausgerichtet. Er baut nicht auf der Intel-Notation, sondern auf der AT&T-Notation auf. Die zu berücksichtigenden Unterschiede der Notationen können der GAS-Dokumentation entnommen werden.

9.3.2 Besondere Datentypen

Alle mehrbytingen Integer-Zahlen sind grundsätzlich im für Intel üblichen little Endian („lowest Byte first“) Format gespeichert. Alle String-Ausgaben innerhalb des Kernels erfolgen im von IBM erweiterten ASCII-Zeichensatz der IBM-PC-Plattform.

Um eine spätere Portierung des Kernels zu erleichtern, verwendet der Kernel speziell benannte Datentypen, die immer bestimmten Anforderungen genügen müssen, um den Kernel korrekt zum Laufen zu bringen. Diese Datentypen sind:

int8_t	8-bit Ganzzahl mit Vorzeichen („signed integer“)
int16_t	16-bit Ganzzahl mit Vorzeichen („signed integer“)
int32_t	32-bit Ganzzahl mit Vorzeichen („signed integer“)
uint8_t	8-bit Ganzzahl ohne Vorzeichen („unsigned integer“)
uint16_t	16-bit Ganzzahl ohne Vorzeichen („unsigned integer“)
uint32_t	32-bit Ganzzahl ohne Vorzeichen („unsigned integer“)
intptr_t	Vorzeichenbehafte Ganzzahl, die durch (void*) in einen Zeiger konvertiert werden darf
uintptr_t	Vorzeichenlose Ganzzahl, die durch (void*) in einen Zeiger konvertiert werden darf
size_t	Anzahl der Bytes eines Objekt als vorzeichenlose Ganzzahl
bool_t	Ganzzahl, deren Aussage mit dem Wert TRUE wahr und dem Wert FALSE (=0) falsch ist.
bool	Identisch zu bool_t
sid_t	SID eines Subjekts (32-bit Länge, Verwendung als vorzeichenlose Ganzzahl zulässig)
irq_t	Nummer eines Interrupts (32-bit Länge, Verwendung als vorzeichenlose Ganzzahl zulässig)
errno_t	Fehlernummer (32-bit Länge, Verwendung als vorzeichenlose Ganzzahl zulässig)

9.3.3 Häufige Idiome

Umwandlung von Integer in Adresse

Um einen Integer in eine Adresse in der gcc-basierten x86-Implementierung korrekt umzuwandeln, ist das Idiom

Zeiger = (void*)(uintptr_t)Integer

zu verwenden.

Thread und Prozesstabelleneinträge

Die Thread und die Prozesstabelle sind als Vektoren des Typs uint32_t* umgesetzt worden. Um die Adresse eines Threaddeskriptors leichter zu ermitteln, wurden die Makros

THREAD(*SID des Threads*, *Nr. des Auszuwählenden Eintrags des Deskriptors*)

PROCESS(*SID des Threads*, *Nr. des Auszuwählenden Eintrags des Deskriptors*)

eingeführt. Mit diesen Makros kann mit Hilfe der SID eines Threads und der Nummer des Eintrags innerhalb des Deskriptors (die verschiedenen Eintragsnummern sind über die THRTAB und PRCTAB Makros erhältlich) auf einen Thread-Deskriptor zugegriffen werden.

Gelegentlich wird aus Gründen der besseren Lesbarkeit oder aus programmiertechnischen Gründen statt dieser Makros direkt auf einen Deskriptor über einen Zeiger zugegriffen. Diese Zeiger werden wie folgt Initialisiert:

Thread_Zeiger = THREAD(*SID des Threads*, 0);

Prozess_Zeiger = PROCESS(*SID des Threads*, 0);

Innerhalb dieser Deskriptoren wird dann auf die Daten wie folgt zugegriffen:

Thread_Zeiger[*Nr. des Auszuwählenden Eintrags des Deskriptors*]

Prozess_Zeiger[*Nr. des Auszuwählenden Eintrags des Deskriptors*]

Eine besondere Rolle spielen die Zeiger *current_t* und *current_p* die jeweils auf den Deskriptor des aktuellen Threads bzw. Prozesses zeigen.

9.3.4 Fehlerbehandlung innerhalb des Quelltextes

Innerhalb des Kernels können während der Arbeit Fehler auftreten. Wird ein Fehler nicht durch einen falschen Parameter eines Systemaufrufs direkt oder indirekt hervorgerufen, so ist der Kernel mit einer Fehlerbotschaft anzuhalten.

Tritt ein Fehler auf Grund eines falschen Parameters eines Systemaufrufs auf, so ist dessen Fehlernummer durch das Makro *SET_ERROR(<Fehlernummer>)* zu setzen. Dieses Makro ändern die Variable *sysc_error*. Der Inhalt dieser Variable wird bei der Rückkehr aus dem Systemaufruf durch den Kernel an den aufrufenden Therad übermittelt.

9.3.5 Inlines, Makros und Inline-Assembler

Im Kernel werden sowohl Inline-Funktionen, wie auch Makros, wie auch Inline-Assembler verwendet. Grundsätzlich sollten komplizierte Makro-Verschachtelungen vermieden werden, um den Quelltext leserlich zu halten. Die Mischung mit dem Inline-Assembler sollte nur dann stattfinden, wenn dies aus Performance-Gründen oder aus Gründen der Leserlichkeit sinnvoll ist.

9.3.6 Benennung von Symbolen

Alle globalen Symbole innerhalb des Kernels sind mit einem Sinngebenden Prefix zu versehen. Derzeit sind folgende Prefices vorgesehen:

i386_	In Assembler implementierte Low-Level-Funktion

sysc_	Implementierung eines Systemaufrufs
kmem_	Symbol das zur Speicherverwaltung gehört
kio_	Symbol das zur I/O-Leitung gehört
ksubj_	Symbol das zur Subjekt-Verwaltung (Prozesse/Threads) gehört
ksched_	Symbol das zum Scheduler gehört
kinfo_	Symbol das zur Systeminformation- oder zur Deskriptorverwaltung gehört
ksync_	Symbol das zum *sync*-Mechanismus gehötr
kremote_	Symbol das zur Threadfernsteuerung gehört

Eine Ausnahme gilt für globale Variablen, die ohne Prefix deklariert werden können.

Makros haben als Prefix meistens ein Prefix, dass sie zu ihrem Einsatzbereich zuordnet, so ist THRTAB_ z.B. eine Zuordnung zur Thread-Tabelle oder THRSTAT eine Zuordnung zu den Thread-Statusflags. Ausnahme bilden Makros, die zur Vereinfachung von Berechnungen dienen - diese Tragen Prefices ihres Einsatzbereiches (abgesehen von den Makros PROCESS und THREAD).

9.3.7 Kommentierung des Quelltextes

In der HydrixOS-Entwicklermailingliste (*hydrixos-internal@hydrixos.org*) wurde beschlossen, dass der Quelltext des Systems durchgehend in englischer Sprache kommentiert wird. In den Quelltext-Kommentar sind Beschreibungen über den Zweck größerer Arbeitsabschnitte einer Funktion aufzunehmen, sofern diese nicht trivial sind.

Ferner ist jede Funktionsdefinition mit einem darüberstehenden Kommentar zu versehen, der folgenden Aufbau haben muss:

```
/*
 * <Funktionsname>(<Parameterliste>)
*
 * <Funktionsbeschreibung, die auf Parameterliste Bezug nimmt>
 *
 *<Beschreibung der möglichen Rückgabewerte>
 *
 * <Hinweise zur Implementierung>
 *
 */
```

Die Parameterliste ist zur besseren Lesbarkeit ohne Datentypen auszuführen, zumal diese unterhalb des Kommentars ohnehin in der Funktionsdeklaration stehen. Nur bei Systemaufrufen sollte die Parameterliste seperat erklärt werden. Ansonsten reicht es aus, wenn die Funktionsbeschreibung auf die Parameter eingeht. Wenn zu viele Parameter vorkommen, sollte sie die in der Parameterliste verwendeten Namen eindeutig aufgreifen und diese in ’einfachen Anführungszeichen’ enthalten.

Der Quelltext soll keine Beschreibung abstrakter und übergeordneter Vorgänge enthalten (z.B. wie ein Thread gewechselt wird), sondern nur lokal den jeweiligen Vorgang des folgenden Codeabschnitts beschreiben. Übergeordnete Beschreibungen sind in diesem Dokument extern dokumentiert.

Eine gute Dokumentation ist die Grundlage der Verstehbarkeit des Projektes. Freie Software soll es dem Benutzer ermöglichen in die Arbeit anderer Programmierer einsicht zu erhalten. Als praktisches Resultat soll er dabei selbst Fehler entdecken können. Dies und die allgemeine Weiterverwendung des Quelltextes ist nur mit einer guten Dokumentation möglich, die auch gemäß den Richtlinien

durchgeführt wird. Wir bitten daher jeden Mitprogrammierer und besonders jeden späteren Maintainer eines HydrixOS-Projektes sich daran zu halten!

9.3.8 Die Quelltextmodule und Header-Dateien

Der Quelltext ist in verschiedene Module und Header-Dateien getrennt. Derzeit liegt der gesamte Quelltext in 'src/hymk/x86'. Bei späteren Portierung kann es sinnvoll sein, den portablen Quelltext in ein getrenntes Verzeichnis unterzubringen. Die internen Header des *hymk* liegen im Verzeichnis 'src/hymk/include'. Sie sollten nicht mit den restlichen Headern des HydrixOS-Projektes vermischt werden, da sie nur intern im Quelltext des *hymk* Sinn ergeben. Der hymk verwendet ferner einige globale Header des HydrixOS-Projektes.

Zur Übersicht der vorhandenen Dateien:

src/hymk/ **Hauptverzeichnis des hymk-Quelltextes**

Makefile	Makefile des hymk
kmap	Symbol-Map, die beim Linken des hymks entsteht

include/ **Include-Dateien des hymk**

x86/ **Include-Dateien der x86-Implementierung**

current.h	Header für Daten und Funktion hinsichtlich des aktuellen Betriebszustands
error.h	Header zur Fehlerbehandlung innerhalb von Systemaufrufen
info.h	Header zur Systeminformation und Deskriptorverwaltung
kcon.h	Header für interne Debug-Konsole
mem.h	Header für Speicherverwaltungsoperationen
page.h	Header für Seitenbezogene Speicherverwaltungsoperationen
sched.h	Header für Scheduling- und Kontextwechseloperationen
setup.h	Feineinstellungen des Kernels
stdarg.h	Abgewandelter ISO-C Header *stdarg.h* (Debug-Konsole)
stdio.h	Abgewandelter ISO-C Header *stdio.h* (Debug-Konsole)
string.h	Abgewandelter ISO-C Header *string.h*
sysc.h	Header für die Implementierung der Systemaufrufe

x86/ **Quelltext-Dateien des hymk**

alloc.c	Funktionen zur Allokation von Seitenrahmen
current.c	Funktionen für die Änderung des aktuellen Arbeitszustandes (insb. Thread- und Prozesswechsel)
frame.c	Funktionen zur Verwaltung von Kernel-Stacks
info.c	Funktionen zur Deskriptor-Verwaltung und Verwaltung der Info-Pages
init.c	Initialisierung des Systemkerns

intr.c	Initialisierung des PICs; Lowlevel-Behandlung aller Software- und Hardwareinterrupts
io.c	I/O-Systemaufrufe und Übergang zur Usermode-IRQ-Behandlung
irq.s	Lowlevel-Behandlung von IRQs und Exceptions; Rückkehr aus Kernel-Mode; Lowlevel-Implementierung des Thread-Wechsels
kprintf.c	Kernel-Debugkonsole
map.c	Implementierung der Shared-Memory-Systemaufrufe
meminit.c	Initialisierung der Kernel-Speicherverwaltungstabellen
modules.c	Verwaltung der von GRUB geladenen Zusatzmodule; Verwaltung der GRUB-Startdaten
page.c	Verwaltung von Seitenrahmen und virtuellen Adressräumen
paged.c	Kommunikation mit dem Paging-Dämon
remote.c	Systemaufrufe für Thread-Fernsteuerung und interne Operationen zur Softint-Umleitung (für *recv_softints*)
schedule.c	Verwaltung der Run-Queue und Scheduler-bezogene Systemaufrufe
security.c	Systemaufruf zum Wechsel in den Root-Modus
start.s	Startup-Code des Kernels (wird von GRUB nach Ladens des Kernels aufgerufen)
string.c	Abgewandelte ISO-C String-Funktionen
subject.c	Systemaufrufe zur Verwaltung von Subjekten (Threads und Prozesse)
sync.c	*sync*-Mechanismus des Kernels und *sync*-Systemaufruf
sysc.s	Lowlevel-Eintrittscode für die Systemaufrufe
timeout.c	Hilfsfunktionen für die Umsetzung zeitbasierter Timeouts (u.a. für *sync* und *recv_softints*)
tss.c	Initialisierung des Task-Status-Segments des Kernels

Der Kernel verwendet zusätzlich folgende Standard-Header des HydrixOS-Projektes

src/include/hymk Projektglobale Header zum hymk

sysinfo.h	Makros für die Thread- und Prozesstabelleneinträge
x86-io.h	x86-IO-Instruktionen als Makros

src/include/hydrixos HydrixOS-Header

errno.h	Makros für Fehlernummern
sid.h	Makros für SID-Datentyp
types.h	Standard-Datentypen innerhalb des HydrixOS-Projektes

9.3.9 Kernel-Debugging

Bildschirmausgabe über kprintf

Um den Kernel-Quellcode debuggen zu können, gibt es eine interne Funktion *kprintf*, die auf den Textbildschirm der IBM-PC-Architektur Zeichen ausgibt. Diese Zeichen sind weiß hervorgehoben und haben am Zeilenanfang ein rotes „>“-Zeichen, um sie leichter dem Kernel zuordnen zu können. Die Kernel-Debugkonsole sollte ausschließlich zum Debuggen des Kernels verwendet werden.

Debug-Modus

Der Kernel kann im Debug-Modus kompiliert werden - dort führen bestimmte Fehlerbehandlungen zum Stopp des Systems und werden verschiedene Ausgaben über die Debug-Konsole gemacht. Dieser Modus wird aktiviert, indem das Makro *DEBUG_MODE* in der Datei setup.h gesetzt wird. Alle Ausgaben, die im Debug-Modus des Kernels erscheinen sollen, sollten wie folgt implementiert werden:

```
#ifdef DEBUG_MODE
kprintf(AUSGABE);
#endif
```

Soll bestimmter Code bei in besonders extensiven Debugging ausgeführt werden, so ist das Makro *STRONG_DEBUG_MODE* zu verwenden. Ist dieses Flag gesetzt, so führen z.B. Exceptions im Benutzermodus zum sofortigen Stillstand des Systems.

Boot-Modus

Der Boot-Zustand des Systems kann über die Variable *kdebug_no_boot_mode* (*init.c, kcon.h*) festgestellt werden. Ist sie wahr (1), so hat der Kernel den Boot-Modus verlassen und das erste Programm gestartet. Ist sie falsch, so befindet sich der Kernel noch in der Initialisierung. Mit Hilfe dieser Variable können u.U. bestimmte Ausgaben während des Ladens des Kernels zu Debugging-Zwecken dargestellt werden, die später im Betrieb des Systems unterdrückt werden sollen.

9.4 Kompiler-Parameter

Der Quelltext von HydrixOS erlaubt verschiedene Feineinstellungen, die über Makros in der Datei *setup.h* geregelt werden können. Diese sind derzeit:

Makro-Name	Standard-Wert	Bedeutung
DEBUG_MODE	(inaktiv)	Der Kernel soll in den Debug-Modus kompiliert werden.
STRONG_DEBUG_MODE	(inaktiv)	Der Kernel soll im besonderen Debug-Modus kompiliert werden. Exceptions im Benutzerodus führen dabei zum Stillstand des Systems!
HYMK_VERSION_MAJOR	N/A	Major-Version des Kernels
HYMK_VERSION_MINOR	N/A	Minor-Version des Kernels
HYMK_VERSION_REVISION	N/A	Revisionsnummer des Kernels
HYMK_VERSION_CPUID	0x80586	Wert des Erkennungs-Code für die Hauptinfopage, den der Kernel für die akteulle Plattform verwenden soll
KERNEL_STACK_PAGES	2	Anzahl der Speicherseiten die ein Kernel-Stacks benötigt
KERNEL_STACK_SIZE	8192	Größe eines Kernel-Stacks in Byte

Makro-Name	Standard-Wert	Bedeutung
ENABLE_PGE_EXTENSION	(aktiv)	Gibt an, ob die Paging-Global-Extension aktiv sein soll oder nicht (dies ist für alle neueren x86-CPUs zu empfehlen)
SCHED_PRIORITY_MAX	40	Maximale statische Priorität des Schedulers (muss größer als die Minimale sein)
SCHED_PRIORITY_MIN	0	Minimale statische Priorität des Schedulers (muss kleiner als die Maximale sein)
SCHED_CLASS_MAX	0	Maximale Scheduling-Klassennummer
SCHED_CLASS_MIN	0	Minimale Scheduling-Klassennummer
TIMER_FREQUENCY	100000	Timer-Frequenz in Hz
IRQ_THREAD_PRIORITY	1000	Priorität, die ein Thread haben soll, wenn er in den IRQ-Behandlungsmodus wechselt

Kapitel 10

Speicherverwaltung

10.1 Aufbau des physikalischen Adressraums

10.1.1 Problematik Normal und High Zone

Da aus Platzgründen nur die unteren 896 MiB des physischen RAMs in den virtuellen Adressraums des Kernels einblendet werden können (siehe Übersicht in Kapitel 6.3), kann er für alle internen Operationen auf Speicherseiten jenseits der 896 MiB-Grenze nicht zugreifen. Aus diesem Grund müssen alle Speicherseiten, die der Kern für eigene Aufgaben verwendet unter dieser Adressgrenze liegen. Für Benutzerprogramme ist diese Grenze unwichtig, da deren Adressraum aus beliebigen Speicherseiten zusammengemappt werden kann.

Der Speicher wird daher grundsätzlich in eine Normal und eine High-Zone eingeteilt. Die Normal-Zone umfasst alle frei zur Verfügung stehenden Seietnrahmen unterhalb der physischen Adresse von 896 MiB, während die High-Zone alle Seitenrahmen jenseits dieser Adressen umfasst. Der Zugriff auf Seitenrahmen der High-Zone erfordert, dass der Kernel diese Seitenrahmen in seinen Teil des virtuellen Adressraums hinein abbildet.

10.1.2 Grundsätzliche Gliederung des physischen Speichers

Grundsätzlich gliedert sich der physikalische Adressraum in der x86-Implementierung in drei Bereiche:

- *Den untere Bereich*, der Kernel-Code und Daten enthält, sowie x86-Spezifische Hardware-Speicherbereiche und die Images der verschiedenen Boot-Server des HydrixOS-Betriebssystems. Der untere Teil beginnt am Anfang des physikalischen Adressraums.
- *Den mittlere Bereich*, der die Seitenrahmen (und erforderlichen Verwaltungstabellen) enthält, die von *alloc_pages* verwendet werden. Wobei der Kernel direkt nur die ersten 894 MiB des Bereichs verwenden kann, was bei internen Allokationen des Kernels beachtet werden muss (im wesentlichen sind davon Page-Tables und Page-Directories betroffen).
- *Den oberen Bereich*, der weitere Hardware-Speicherbereiche enthalten kann und vollständig für io_alloc freigegeben ist. Dieser Bereich fängt direkt nach dem *mittleren Bereich* an und endet an der oberen Grenze des physikalischen Adressraums (4 GiB).

10.1.3 Gliederung des unteren Bereichs

Der untere Bereich des physikalischen Adressraum hat eine statische Größe von 2 MiB und ist folgende Unterbereiche gegliedert:

Adressbereich	Bedeutung
`0x00000000 - 0x0000FFFF`	BIOS-Daten (freigegeben für io_alloc)
`0x00010000 - 0x000B7FFF`	• Initialer Kernel-Stack bei `0x00090000` (verfällt nach Initialisierung des Kernels; sollte nicht unter `0x00080000` fallen) • Nach Initialisierung des Systems kann dieser Bereich mittels io_alloc() verwendet werden
`0x000B8000 - 0x000FFFFF`	EGA / VGA-Speicher; BIOS-ROM (freigegeben für io_alloc)
`0x00100000 - 0x001FFFFF`	• Image des Kernels • Image des Init-Servers • Images weiterer Server (freigegeben für io_alloc)

10.1.4 Gliederung des mittleren Bereichs

Der mittlere Bereichs des physikalischen Adressraums besitzt hingegen keine statische Größe und ist abhängig von der allgemeinen Größe des eingebauten RAMs (die Größe wird beim Start des Systems durch GRUB ermittelt). Er gliedert sich in folgende Bereiche (die Variable *R* ist die Größe des kompletten verfügbaren RAMs, *N* die Größe der kompletten Normal Zone, *H* die Größe der kompletten High Zone - jeweils in Byte):

Name	Größe	Bedeutung
Initial Kernel Page Tables (IKP)	1 MiB + 4 KiB	Enthält die initale Page Table und das Page Directory des Kernels. Die Page Tabels dieses Adressraums werden von allen anderen virtuellen Adressräumen mitbenutzt, um den Kerneladressraum abzubilden. Die IKB ist entlang einer Seitengrenze angepasst.

Name	Größe	Bedeutung
Page Buffer Table (PBT)	(*R* / 4096) * 14	Enthält eine Tabelle aus Einträgen zu je 4 Byte. Diese Tabelle beinhaltet einerseits einen Zähler, der angibt, für wieviele virtuelle Seiten ein Seitenrahmen des RAM derzeit verwendet wird (die Tabelle verwaltet alle Seiten ab der 1-MiB-Grenze). Ist der Zähler gleich 0, ist der Rahmen derzeit nicht verwendet. Ist der Zähler gleich 1, wird er von einem Prozess an einer bestimmten Stelle verwendet. Die weiteren Byte des Eintrags beinhalten dann die SID des Prozesses und die Adresse, an welcher der Seitenrahmen im virtuellen Adressraum des Prozesses verwendet wird. Ist der Zähler größer 1, wird die Seite möglicherweise von verschiedenen Prozessen verwendet und ist an verschiedenen virtuellen Seiten eingeblendet. Daher verweisen die folgenden Byte des Eintrags auf eine Liste aus Einträgen der PST, mit deren Hilfe herausgefunden kann, wo der Seitenrahmen derzeit verwendet wird.
Page Share Table (PST)	((*R* / 4096) * 0.2) * 12	Die PST ist keine konsistente Tabelle. Sie ist vielmehr eine Puffer, indem verkettete Listen angelegt werden können. Im Grundzustand besteht in der PST nur eine verkettete Liste, die alle ungenutzten Einträge der PST miteinander verkettet, so dass rasch ein neuer Eintrag der PST alloziiert werden kann. Eine PST-Liste besteht aus Einträgen, die angeben, welcher Prozess (SID) an welcher virtuellen Adresse einen Seitenrahmen verwendet. Jeder Eintrag ist mit dem nächsten Eintrag dieser Liste verknüpft. Die Liste selbst ist einem PBT-Eintrag zugeordnet, der zu einem Seitenrahmen gehört, der an mehr als einer virtuellen Adresse eingeblendet ist. Die PST ist so ausgelegt, dass für 20% aller Speicherseiten (beider Zonen) jeweils ein Eintrag in ihr reserviert ist. Eine dynamische Lösung könnte in späteren Versionen des *hymk* folgen.
Normal Zone Free Memory Stack (NZS)	(*N* / 4096) * 4	Der Normal Zone FMS ist ein LIFO-Stack, indem die physikalischen Adressen aller derzeit freier Seitenrahmen der Normal Zone abgespeichert sind, sofern sie aus dem *Page Buffer* stammen.

Name	Größe	Bedeutung
High Zone Free Memory Stack (HZS)	(*H* / 4096) * 4	Der High Zone FMS ist ein LIFO-Stack, indem die physikalischen Adressen aller derzeit freier Seitenrahmen der High Zone abgespeichert sind, sofern sie aus dem *Page Buffer* stammen.
Page Buffer	*R* - (*PBT* - *PST* - *NZS* - *HZS*)	Enthält die Seitenrahmen (wobei manche Kernel-Datenstrukturen in den ersten 896 MiB des RAMs liegen müssen). Der Pagebufferr ist entlang einer Seitengrenze angepasst.

10.2 Organisation des virtuellen Adressraums

Zur „Konstruktion“ des virtuellen Adressraums verwendet der *hymk* zwei plattformspezifische Mechanismen der x86-Architektur für Segmentierung und Paging.

Die Segmentierung setzt bei der x86-Architektur auf den durch das Paging gebildeten linearen Adressraum auf. Darüber ob nun dieser lineare Adressraum dem physikalischen Adressraum entspricht oder ein durch das Paging gebildeter virtueller Adressraum ist, gibt der Begriff „linearer Adressraum“ keine Auskunft. Der Begriff wird in diesem Dokument nur verwendet, um vom segmentierten Adressraum zu unterscheiden (dessen Segmente zwar intern auch linear sind, aber nur über eine Kombination von Segment und Offsetadress adressiert werden können und somit von außen nicht mehr linear adressiert werden können). Ist der physikalische (und auch lineare) Adressraum gemeint, so wird dies explizit genannt - ebenso, wenn speziell ein linearer, virtueller Adressraum gemeint ist.

10.2.1 Einsatz der Segmentierung

Die Segmentierung wird im wesentlichen nur dazu eingesetzt, um den Kernel-Code zu reloziieren. Nach dem Start des Systems wird der Kernel an die physikalische Speicheradresse 1 MiB geladen. Wie aber aus Kapitel 6.3 entnommen werden kann, wird im Betrieb des Systems der Adressbereich des Kernels (der identisch mit den ersten 896 MiB des phyiskalischen Adressraums ist) im virtuellen Adressraum an Adressen > 3 GiB verlagert. Um die Implementierung einfach zu halten, wird für innerhalb des Kernel-Codes verwendete Adressen für direkte Speicherzugriffe auf den physikalischen Adressraum der ersten 896 MiB intern kein Offset verwendet und der Kernel-Code an die Adresse 1 MiB gelinkt.

In der x86-Architektur werden alle Segmente durch sog. Selektoren referenziert. Ein Selektor verweist mehr oder weniger auf einen Eintrag in der sog. globalen Deskriptorentabelle (global descriptor table, GDT). Dieser GDT-Eintrag beschreibt wiederum nebst verschiedenen Zugriffsrechten die Startadresse und die Größe des Segments. Immer wenn ein Programm auf der x86-Architektur auf den Speicher zugreift, muss es dies über eine Adresse tun, die einen Segmentselektor enthält, der das Segment referenziert und ein Offset innerhalb des Segments definiert. Der Segmentselektor wird meist implizit über eines der Segmentregister übergeben, wobei der hymk hier nur auf das Code-Segmentregister (CS) für Code-Zugriffe, das Daten-Segmentregister (DS) für Datenzugriffe und das Stack-Segmentregister (SS) für Stackzugriffe zurückgreift. Die anderen Segmentregister werden höchstens in Ausnahmefällen verwendet. Das GS-Segmentregister ist innerhalb des Kernels normalerweise auf das Datensegment des Benutzerprozesses gestellt, damit Speicherseiten aus diesem Bereich mittels *INVLPG* invalidiert werden können.

Wird nun auf den Speicher zugegriffen, so addiert die CPU die übergebene Offsetadresse auf die Startadresse des Segments und verwendet diese lineare Adresse, um auf den linearen Adressraum zuzugreifen. Zusätzlich ermöglichen Zugriffsrechte die Einschränkung der Verwendung von Segmenten

für den Kernel- und den Benutzer-Modus über den sog CPL-Wert des Segmentdeskriptors (weitere Details können der IA32-Dokumentation von Intel entnommen werden).

Der Kernel verwendet insgesamt sieben Deskriptoren, wobei im wesentlichen davon nur vier verwendet werden. Diese Deskriptoren sind:

Sel.	Start	Größe	Zugriff	Zweck
0x00	0	0	Keiner	Dummy-Deskriptor; von x86-Architektur vorgeschrieben; wird aber nicht verwendet
0x08	0x00000000	4 GiB	Nur Kernel	Initiales Code-Segment, das den gesamten linearen Adressraum umfasst
0x10	0x00000000	4 GiB	Nur Kernel	Initiales Daten/Stack-Segment, das den gesamten linearen Adressraum umfasst
0x18	0xC0000000 (~ 3 GiB)	1 GiB	Nur Kernel	Kernel-Code Segment
0x20	0xC0000000	1 GiB	Nur Kernel	Kernel-Daten/Stack-Segment
0x28	0x00000000	4 GiB	Jeder	User-Code-Segment
0x30	0x00000000	4 GiB	Jeder	User-Daten/Stack-Segment
0x38	(Dynamisch)	100 B	Kernel	Task-Status-Segment (von x86-Architektur vorgeschrieben)

Die GDT wird statisch durch das Symbol *i386_gdt_s* (*start.s*) definiert. Ihre nähere Beschreibung kann der IA-32-Dokumentation und der Quelltextkommentierung entnommen werden. Zum Laden der GDT muss ein Dummy-Deskriptor verwendet werden (siehe beschreibung der LGDT-Instruktion). Hierfür gibt es zwei Deskriptoren - einen, der vor (*i386_gdt_des, start.s*) und einen der nach (*i386_gdt_des_new, start.s*) Relokation des Kernels verwendet wird.

Auffällig ist, dass es ein initales Code- und Datensegment, sowie ein „normales“ Code- und Datensegment für den Kernel gibt. Dies hängt ebenfalls mit der Relokation des Kernels zusammen - bevor der Kernel im linearen Adressraum neu plaziert wird, benötigt er eine Zugriffsmöglichkeit auf den vollständigen linearen Adressraum (der zu diesem Zeitpunkt noch dem physikalischen entspricht). Dafür wird ein Segment angelegt, dass den linearen Adressraum komplett abbildet. Ist der Kernel im linearen Adressraum reloziiert worden, wird die neue, reloziierte IDT mit dem Deskriptor *i386_gdt_des_new, start.s* und die neuen Code- und Datensegment (0x18 bzw. 0x20) geladen. Ab diesem Zeitpunkt kann der Kernel nur noch auf die unteren 896 MiB zugreifen.

Ebenfalls auffällig ist, dass der User-Code und die User-Daten sich mit dem linearen Adressbereich des Kernels überscheindet. Da hier der im Benutzermodus ausgeführte Code Zugriffsrecht hat, müsste ja zu befürchten sein, dass ein ungültiger Zugriff auf den Kernel-Speicher möglich ist. Dies wird jedoch auf Ebene des Pagings geregelt. Die Überschneidung ist notwendig, damit der ausgeführte Code im Benutzermodus Zugriff auf die Info-Pages erhalten kann.

10.2.2 Einsatz des Pagings

Um einen virtuellen, linearen Adressraum zu bilden, verwendet der Kernel die Paging-Fähigkeiten der x86-Architektur. Diese bietet ein 2-Stufiges Paging-Verfahren an. Die 1. Stufe bildet die sog. Page Directory, deren 1024 Einträge jeweils die physikalische Adresse einer Page Table enthält. Die 1024 Einträge einer Page Table wiederum enthalten die physikalischen Adressen der Seitenrahmen, die für die jeweilige virtuelle Speicherseite verwendet werden soll. Mit 1024 Seiten die je 4096 Byte groß sind, die in 1024 Seitentabellen umschrieben werden, können folglich 4 GiB umschrieben werden

(1024 * 1024 * 4096 Byte = 2^32 Byte = 4 GiB). Alle Page-Table und Page-Directory-Einträge haben den Typ *uint32_t*.

Die Einträge in der Page Table enthalten zudem neben der Adresse des Seitenrahmes verschiedene Zugriffsrechte, die für die jeweilige Seite im virtuellen Adressraum gelten werden. Die Zugriffsrechte in den Einträgen der Page Directory wirken sich übergeordnet auf die Zugriffsrechte der jeweiligen Seite aus.

Die Zugriffsrechte der x86-Architektur werden im Kernel-Code durch die *PFLAG_*-Makros (*mem.h*) abgebildet. Intern sollten jedoch im Code die *GENFLAG_*-Makros (*mem.h*) verwendet werden, da diese die plattformspezifischen Flags mit plattformunabhängigen Begriffe abbilden (dies soll spätere Portierungen erleichtern), da die x86-Architektur in ihrer Flexibilität bei den Seitenzugriffsrechten manchen Architekturen in eingem Nachsteht. Die Bedeutung der einzelnen *PFLAG_*-Makros bzw. *GENFLAG_*-Makros sollte sich aus deren Namen und anhand der IA-32-Dokumentation erschließen lassen. Einige der *GENFLAG_*-Makros sind nicht Deckungsgleich mit den IA-32-Flags. Das Flag *GENFLAG_EXECUTABLE* hat kein IA-32-Gegenstück, existiert aber trotzdem, um eine nachträgliche Portierung des Quellcodes zu erleichtern. Das Flag *GENFLAG_DONT_OVERWRITE_SETTINGS* ist kein Page-Flag, sondern dient lediglich zur Informierung der internen Page-Mapping-Routine und wird später noch erläutert.

Das Flag *GENFLAG_DO_COPY_ON_WRITE* hat keine unmittelbare Hardware-Funktion. Es wird über das Flag *PFLAG_AVAILABLE_0*, das dem Systemprogrammierer zur freien Verfügung steht, umgesetzt. Ist das Flag gesetzt, so soll auf diese Seite die Copy-On-Write-Operation ausgeführt werden. Das Flag *GENFLAG_PAGED_PROTECTED* hat auch keine Hardware-Entsprechung. Dieses Flag löst bei einer *unmap*-Operation, die nicht durch den Paging-Dämon ausgeführt wird, eine Exception aus, so dass der Paging-Dämon die Seite gesondert behandeln kann. Es wird effektiv über das Flag *PFLAG_AVAILABLE_1* gespeichert.

Nach der Initialisierung der Paging-Speicherverwaltung kann die Adresse der aktuell verwendeten Page-Directory dem Zeiger *i386_current_pdir* (*mem.h*). Die Page-Tables, die den Kernel-Adressraum umschreiben (und somit in jeder Page-Directory konstant vorhanden sind), liegen im IKP-Bereich des physikalischen Adressraums (siehe 10.1.4) und ist an eine entsprechende Adresse im Kerneladressraum (also Originaladresse + 3 GiB) eingeblendet. Im Kernel-Code kann auf diesen Bereich über den Zeiger *ikp_start* (*mem.h*) zugegriffen werden. Die ersten 4096 Byte des IKP-Bereichs enthalten die ursprüngliche Kernel-Pagedirectory, die als Vorlage für neue Adressräume verwendet werden kann. Die restlichen 1 MiB der IKP enthalten die Seitendeskriptoren der oberen 1 GiB des virtuellen 4-GiB-Adressraums. Die Verwendung der Deskriptoren kann der Beschreibung des oberen GiB des virtuellen Adressraums in Kapitel 6.3 entnommen werden.

10.3 Verwaltung von Seitenrahmen

10.3.1 Verwaltung freier Seitenrahmen

Die Hauptaufgabe einer Speicherverwaltung ist natürlich die Reservierung und Freigabe von Speicher. Die Speicherverwaltung des *hymk* wurde gezielt einfach gehalten. Sie verzichtet - im Unterschied zu vielen monolitischen Systemen - auf unterschiedliche Allokationswege für verschiedene Anforderungsarten (kleine Speicherbereiche etc.), sondern bietet nur einen Mechanismus zur Allokation von kompletten Seitenrahmen aus dem physikalischen Adressraum. Zusätzlich bietet sie noch einen Kontrollmechanismus bei der Speicherfreigabe, bzw. beim Seitenmapping, der eine möglichst übersichtliche und für einen Paging-Dienst transparente Mehrfachnutzung von Seitenrahmen ermöglicht.

Die Allokationsstrategie der Speicherverwaltung ist recht schlicht gehalten. Sie baut im wesentlich auf einem Stack nach dem LIFO-Prinzip auf, dessen Einträge jeweils immer die physikalische Adresse eines freien Seitenrahmens enthält. Diese Art Stack wird im *hymk* als Free Memory Stack

(FMS) bezeichnet. Auf Grund der begrenzten Größe des Kerneladressraums benötigt der *hymk* für die x86-Plattform zwei Stacks jeweils für die *Normal* und die *High-Zone* (in späteren Versionen könnte ebenfalls ein dritter Stack für den DMA-Controller erforderlich sein.).

Seitenrahmen, die für den Kernel bestimmt sind können nur aus dem Stack der *Normal Zone* bezogen werden, während Benutzerprogramme aus beiden Stacks ihre Seitenrahmen beziehen können. Um so viel wie möglich an Normal-Zone-Speicher freizuhalten, werden - wenn möglich - für Allokationen aus dem Benutzeradressraum zuerst Seiten aus der *High-Zone* verwendet und erst nach Verbrauch dieser die *Normal-Zone* in Angriff genommen.

Wird ein freier Seitenrahmen angefordert wird von einem dieser FMS jeweils ein Eintrag vom Stack heruntergeladen und der Stackpointer inkrementiert. Wird hingegen ein Seitenrahmen wieder freigegeben, wird der Stackpointer dekrementiert und an dessen aktuelle Adresse dann die Adresse des neuen Seitenrahmens geschrieben. Vorteil dieses Verfahrens ist seine Einfachkeit - allerdings auf Kosten einer möglicherweisen hohen externen Fragmentierung (d.h. es kann nicht garantiert werden, dass Seitenrahmen, die in einem virtuellen Adressraum am Stück liegen auch tatsächlich im physikalischen Speicher am Stück gespeichert sind). Außerdem können niemals mehr als eine Speicherseite zusammenhängend alloziiert werden. Dies stellt allerdings kein größeres Problem dar, da der *hymk* ohnehin keine nicht fest vorgegebenen Datenstrukturen besitzt, die größer als eine Speicherseite sind und für Benutzerprogramme der physikalische Zusammenhang von Speicherseiten weniger Bedeutung trägt. Allerdings bedeutet dies auch, dass für die Alloziierung größerer Speichermängen viele kleine Einzelschritte erforderlich sind (wobei die Tatsache mehr ins Gewicht fällt, dass zur Erfüllung von gängigen Sicherheitsstandards alle Seitenrahmen nach der Alloziierung mit dem Wert 0 aufgefüllt werden). Ob sich dieses Verfahren bewährt, wird sich noch zeigen.

Da neben dem Alloziieren auch das Einblenden der Speicherseiten in den Benutzeradressraum Zeit kostet, muss die maximale Anzahl von Seiten, die während eines Systemaufrufs alloziiert werden können, beschränkt werden. Derzeit liegt die Beschränkung bei 8 MiB (2048 Seiten), da diese Speichermenge selbst auf langsameren Rechnern mit wenig Zeitverbrauch möglich ist und nur relativ selten schlagartig alloziiert werden muss.

10.3.2 Verwaltung benutzter Seitenrahmen

Wenn ein freier Seitenrahmen reserviert worden ist, wird er entweder vom Kernel für Speicherzwecke verwendet oder von Prozessen in deren virtuelle Adressräume eingebaut - und möglicherweise als gemeinsamer Speicherbereich mittels *map* über mehrere andere Prozesse hinweg gemeinsam genutzt. Damit sowohl der Kernel, als auch ein späterer Paging-Dienst herausfinden kann, in welchem Prozess und an welcher Adresse ein Seitenrahmen zu finden ist, ist eine „umgekehrte Seitentabelle“ erforderlich, die eben dieses referenziert. Unter HydrixOS erledigen diese Aufgaben die *Page Buffer Table* (PBT) und die *Page Share Table* (PST).

Die PBT enthält für jeden Seitenrahmen des mittleren Bereichs einen Eintrag. Dieser Eintrag beschreibt, ob ein Seitenrahmen bereits genutzt wird und wenn ja, an wie vielen virtuellen Adressen. Wird dieser nur als Seitenrahmen für eine virtuelle Seite verwendet, enthält sie ferner noch ein sog. *SID:VA-Paar* (= SID des Prozesses, der die Seite Verwendet und virtuelle Adresse in dessen Adressraum), über das eine Seite in einem bestimmten virtuellen Adressraum lokalisiert werden kann. Wird der Seitenrahmen hingegen an mehreren virtuellen Adressen und möglicherweise von verschiedenen Prozessen genutzt, verweist sie nur auf eine verkettete Liste aus *SID:VA-Paaren*, die über die PST verteilt abgelegt sind. Dies wird dadurch realisiert, dass ein Element der PBT als Union deklariert ist und jenachdem vom Programmcode als *SID:VA-Paar* oder als Zeiger auf eine solche verkettete Liste gesehen wird. Ein solches *SID:VA-Paar* wird durch die Datenstruktur *page_user_t* (*mem.h*) näher beschrieben:

```
typedef struct {
```

```
    sid_t pid;          /* SID des Prozesses */
    uintptr_t u_adr;    /* Virtuelle Adresse */
}page_user_t;
```

Die PBT selbst ist vom Typ *pbtab_t* (*mem.h*). Das Union, das einerseits aus einem Zeiger auf eine PST-Liste und andererseits Einzelbenutzereinträge enthält ist vom Typ *page_usr_u* (*mem.h*).

```
typedef union {
    page_user_t single;   /* Einzelbenutzer SID:VA */
    sidlist_t *multi;     /* PST-Liste für Mehrbentuzer */
}page_usr_u;
typedef struct {
    uint16_t usage;       /* Anzahl der Verwendungen */
    page_usr_u owner;     /* Benutzer-Union */
}pbtab_t;
```

Das Feld 'usage' gibt an, an wie vielen Stellen der Seitenrahmen nun verwendet wird. '0' bedeutet keiner, '1' bedeutet einer - wobei das *single*-Element des Unions *owner* zu beachten ist, während '>1' bedeutet, dass mehrere den Seitenrahmen benutzen und das *multi*-Element des Unions *owner* beachtet werden muss.

Deklariert ist die PBT durch die Variable *page_buf_table* (*meminit.c*; *mem.h*). Um eine physikalische Adresse in einen PBT-Eintrag leicht umrechnen zu können, existiert das Makro *PAGE_BUF_TAB(___padr)* (*mem.h*). Dieses Makro erhält als Parameter die physikalische Adresse und kann selbst als PBT-Eintrag behandelt werden (z.B. `PAGE_BUF_TAB(addr)->usage++`).

Was sind nun diese PST-Listen? Die PST ist keine zusammenhängende Tabelle, sondern vielmehr ein Puffer für einfach verkettete Listen, deren Elemente jeweils eine feste Struktur haben (bestehend aus einem *SID:VA-Paar*). Eine PST-Liste beschreibt schlichtweg alle Prozesse und alle virtuellen Adressen, an denen ein Seitenrahmen verwendet wird. Dadurch kann ein Paging-Dienst relativ schnell herausfinden, wo er eine Seite entfernen muss, damit er den Seitenrahmen z.B. auf die Festplatte auslagern kann. Die Liste ist nur einfach verkettet, d.h. es kann jeweils nur der Nachbar in vorwärtiger Richtung festgestellt werden. Der Anfang einer solchen Liste ist im *multi*-Element des zum Seitenrahmen gehörenden PBT-Eintrags verzeichnet.

Jedes Element des Puffers hat den Typ *sidlist_t* (*mem.h*):

```
typedef struct sidlist_st{
    page_user_t user; /* SID:VA-Paar */
    struct sidlist_st *nxt; /* Zeiger auf nächsten Eintrag */
}sidlist_t;
```

Alle freien Elemente der PST sind selbst auch als eine verkettete Liste der unbenutzten Einträge verbunden. Der Anfang dieser Liste kann über den Zeiger *page_share_table_free* (*meminit.c*; *mem.h*) in Erfahrung gebracht werden. Immer wenn ein freies Element für eine PST-Liste benötigt wird, wird der erste Eintrag aus der Free-Liste herausgenommen und an den Anfang der anderen PST-Liste eingefügt (dies ist z.B. der Fall, wenn eine Seite durch *map* gemeinsam genutzt werden soll). Wird ein zuvor gebrauchter PST-Eintrag wieder freigegeben, wird er wieder in die Free-Liste eingefügt (dies geschiet z.B. bei *unmap*).

Wenn ein Seitenrahmen von der Einfachebenutzung zur Mehrfachbenutzung wechselt, muss stets eine neue PST-Liste erstellt werden, die sowohl den neuen Mitbenutzer, als auch die Daten des alten Benutzers des Rahmens (aus dem *single*-Eintrag des zugeordneten PBT-Eintrags) enthält. Wechselt er

hingegen von der Mehrfachbenutzung zur Einfachbenutzung, muss die PST-Liste (die nun nur noch einen Eintrag enthalten würde) aufgelöst werden, und der Inhalt des einen PST-Eintrags in das *single*-Element des PBT-Eintrags des Seitenrahmens übernommen werden. Da diese ja als Union organisiert sind, überschreibt der Inhalt des *multi*-Elements das *single*-Element.

Durch dieses Verfahren kann relativ schnell, einfach und trotzdem platzsparend eine Verwaltung des belegten Speichers durchgeführt werden. Nachteil des Verfahrens ist derzeit noch, dass die PST-Liste eben nur aus Einträgen der PST bestehen kann, die ihrerseits eine statische Größe im Speicher ist. Ist die PST voll, können keine weiteren Speicherseiten mehr gemeinsam genutzt werden, selbst wenn noch genügend RAM frei wäre. In späteren Versionen soll hier die Verwaltung der PST selbst dynamisch gemacht werden.

10.3.3 Interne Schnittstellen

Reservierung von Seitenrahmen im Kernel

Intern gibt es zur Reservierung von Speicherseiten zwei Funktionen: *kmem_alloc_user_pageframe* (*alloc.c*; *mem.h*) und *kmem_alloc_kernel_pageframe* (*alloc.c; mem.h*). Beide Funktionen reservieren über die Stacks einen einzelnen Seitenrahmen geben und einen *void*-Zeiger auf die physikalische Adresse des Seitenrahmens zurück. Wird *NULL* zurückgegeben, schlug die Reservierung fehl.

Die Funktion *kmem_alloc_kernel_pageframe* bezieht Seitenrahmen nur aus der *Normal-Zone* und schreibt in den PBT-Eintrag der neu reservierten Seite als Besitzer den SID-Platzhalter *kernel* und als Adresse die Adresse des Seitenrahmens im virtuellen Kerneladressraum (Adresse des Seitenrahmens + 3 GiB) ein, wodurch ein Paging-Dienst erkennen sollte, dass dieser Seitenrahmen dem Kernel gehört und *auf keinen Fall* ausgelagert werden darf. Da diese Funktion die *Normal-Zone* belastet und die entsprechenden Kernel-Markierungen enthält, sollte sie eben nur für Anforderungen des Kernels verwendet werden. Sie sollte daher besser auch nicht durch *map* etc. in Benutzeradressräume gelangen.

Die Funktion *kmem_alloc_user_pageframe* hingegen bezieht Seitenrahmen zuerst aus der *High-Zone*, bis diese aufgebraucht ist (falls sie überhaupt vorhanden war...) und erst dann aus der *Normal-Zone.* Somit können auf diese Seitenrahmen, die diese Funktion liefert nicht vom Kernel direkt zugegriffen werden. Als Parameter erhält diese Funktion das *SID:VA*-Paar, dem der Seitenrahmen später einmal zugeordnet werden soll. Die Funktion trägt diese Daten als Einzelbenutzer in das *single-Element* des PBT-Eintrags des Seitenrahmens ein.

Beide Funktionen greifen auf ein Makro mit Namen *kmem_zero_out* (*alloc.c*) zurück. Dieses Makro füllt den Seitenrahmen mit 0-Bytes auf, so dass frisch alloziierte Seitenrahmen mit 0 initialisiert sind und keine früheren (möglicherweise Sicherheitskritischen) Inhalte mehr enthalten können. Hier ist zu beachten, dass der Kernel bei Allokationen aus der High-Zone zum Überschreiben der alten Seiteninhalte den Seitenrahmen zuerst in einen für ihn zugänglichen Speicherbereich (hierfür ist der UMCA-Bereich vorgesehen, siehe 6.1) mappen muss. Beide Funktionen setzen ebenfalls den Usage-Counter des Seitenrahmens auf 1.

Die Funktion *sysc_alloc_pages* (*alloc.c*) ist die Implementierung des *alloc_pages*-Systemaufruf. Diese sollte neben dem Copy-On-Write-Handler des Kernels (*kmem__on_write, page.c*) als einzige derzeit *kmem_alloc_user_pageframe* verwenden. Die Systemaufrufsimplementierung kann im Kernel intern auch zur Alloziierung und anschließenden Plazierung von Seitenrahmen verwendet werden. Der Systemaufruf wird im folgenden Kapitel über Page Mappings näher erläutert.

Freigabe von Seitenrahmen des Kernels

Seitenrahmen, die zuvor mit *kmem_alloc_kernel_pageframe* reserviert wurden, müssen anschließend durch *kmem_free_kernel_pageframe* (*alloc.c*; *mem.h*) wieder freigegeben werden. Als Parameter erhält diese Funktion einen *void*-Zeiger auf den Seitenrahmen. Diese Funktion überführt den Seiten-

rahmen direkt in den passenden Zonenstack (sowohl Normal, als auch High-Zone, da die Funktion auch von *kmem_free_user_pageframe* verwendet wird). Der PBT-Eintrag wird ungeachtet irgendwelcher verknüpfter PST-Einträge zurückgesetzt - daher sollten mit dieser Funktion direkt nur Kernel-Seitenrahmen freigegeben werden und diese Kernel-Seitenrahmen wiederum niemals mittels *map* zur gemeinsamen Nutzung verwendet werden.

Freigabe von Seitenrahmen aus dem Userland

Seitenrahmen, die hingegen zuvor mit *kmem_alloc_user_pageframe* reserviert wurden, müssen anschließend durch *kmem_free_user_pageframe* (*alloc.c*; *mem.h*) freigegeben werden. Diese Funktion erhält als Parameter das *SID:VA*-Paar, das beschreibt, an welcher Stelle der Seitenrahmen aus einem virtuellen Adressraum entfernt wurde. Die Funktion prüft anschließend, ob es diese Stelle gibt und entfernt diese ggf. aus der PST-Liste des Seitenrahmens. Wurde der Seitenrahmen zuvor nur noch von einem Prozess verwendet, gibt die Funktion den Seitenrahmen entgültig mittels *kmem_free_kernel_pageframe* wieder frei. Wenn andere Prozesse den Seitenrahmen noch verwenden, bleibt er hingegen weiterhin bestehen, nur die Nutzungsinformation wurde entfernt und der *usage*-Zähler des PBT-Eintrags wurde dekrementiert.

10.4 Verwaltung virtueller Adressräume

10.4.1 Erzeugung und Vernichtung virtueller Adressräume

Ein virtueller Adressraum wird immer dann erstellt, wenn ein Prozess den Systemaufruf *create_process* aufruft. Die Behandlungsroutine des Systemaufrufs (*sysc_create_process, subject.c*) wiederum erstellt einen virtuellen Adressraum durch die Funktion *kmem_create_space* (*page.c*). Diese Funktion reserviert zuerst ein Kernel-Pageframe für eine Page-Directory. Die oberen 3 GiB des Page Directories werden mit den entsprechenden Einträgen der IKP verknüpft, so dass der virtuelle Kerneladressraum in den neuen Adressraum eingeblendet wird.

Für die Auflösung eines virtuellen Adressraums ist die interne Funktion *kmem_destroy_space* (*page.c*) zuständig, die als Parameter die Adresse des Page-Directories des Adressraums erhält. Diese Funktion gibt zuerst alle im Page Directory aufgeführten Seitentabellen frei, die unterhalb der 3-GiB-Grenze liegen. Anschließend wird das Page-Directory freigegeben.

10.4.2 Abrufen und Erzeugen von Seitentabellen

Wenn das Mapping von virtuellen Seiten geändert oder abgerufen werden soll, kann es oft passieren, dass zu einer entsprechenden virtuellen Seite noch keine Seitentabelleneintrag existiert. Um dies möglichst transparent zu gestalten, gibt es die Inline-Funktion *kmem_get_table* (*page.h*). Diese Funktion sucht im gegebenen Page Directory nach dem Eintrag der gesuchten Page Table anhand einer übergebenen virtuellen Adresse. Ist keine Page Table vorhanden, wird von dieser Funktion automatisch eine erstellt, falls dieses gewünscht ist. Andernfalls wird ein NULL-Pointer zurückgegeben. Ist die Page Table jedoch vorhanden, wird einfach der Zeiger auf diese zurückgegeben.

10.4.3 Einblenden von Seitenrahmen

Um einen Seitenrahmen an eine virtuelle Adresse einzublenden, muss der Seitendeskriptor (der in einer über *kmem_get_table* erhältlichen Seitentabelle liegt) der zugeordneten virtuellen Seite geändert und dessen Zugriffsrechte und Steuerflags dem Bedarf entsprechend angepasst werden. Handelte es sich zuvor um eine Seite, deren *GENFLAG_GLOBAL*-Flag gesetzt war oder um eine Seite des

derzeit aktiven virtuellen Adressraums, muss anschließend der TLB für diese Seite invalidiert werden, da sonst die CPU möglicherweise noch einen falschen Seitendeskriptor im TLB hält und die virtuelle Seite dann falsch dereferenziert. Da es sich hierbei um eine Assembleranweisung (*INVLPG*) handelt, bietet der Kernel zur Vereinfachung ein gleichnamiges Makro *INVLPG* (*mem.h*) an, dass dieses übersichtlich erledigt. Außerdem muss immer dann, wenn eine aktive Speicherseite (*GENFLAG_AVAILABLE* ist gesetzt) eingeblendet wird, in der Speicherverwaltungstabelle des Prozesses (siehe 8.1.2) der Eintrag, der der jeweiligen Seitentabelle entspricht inkrementiert, bzw. beim Ausblenden dekrementiert werden.

Je nach Fall wird im Kernel das Einblenden der Speicherseiten für die einzelne Funktionalität jeweils separat implementiert oder es wird auf zwei Standardfunktionen zurückgegriffen, die der Kernel anbietet: *kmem_map_page_frame* (*page.c*, *mem.h*) und *kmem_map_page_frame_cur* (*page.c*, *mem.h*). Die erste Funktion muss immer dann verwendet werden, wenn ein Seitenrahmen in einen virtuellen Adressraum eingeblendet werden soll, der nicht dem aktuellen entspricht. Sie darf nicht verwendet werden, wenn der Zieladressraum dem aktuellen entspricht, da sie die TLB-Einträge nicht invalidiert. Die zweite Funktion kann nur für den aktuellen virtuellen Adressraum verwendet werden. *kmem_map_page_frame_cur* greift zwar auf *kmem_map_page_frame* zurück - invalidiert aber im Nachhinein den TLB. Im wesentlichen wird durch beide Funktionen der Seitendeskriptor geändert und der Status in der Speicherverwaltungstabelle des dem Adressraum zugeordneten Prozesses geändert. Wenn als Parameter das Flag *GENFLAG_DONT_OVERWRITE_SETTINGS* übergeben wird, werden bereits bestehende Mappings von beiden Funktionen nicht überschrieben.

10.4.4 Der Systemaufruf *alloc_pages*

Der Systemaufruf *alloc_pages* wird durch die Funktion *sysc_alloc_pages* (*alloc.c*) implementiert. Der Systemaufruf alloziiert - nach Prüfung der übergebenen Parameter - einzelne Seitenrahmen über *kmem_alloc_user_page_frame* (*alloc.c*) und blendet diese in den aktuellen Adressraum mittels *kmem_map_page_* ein. Wichtig ist, dass das *GENFLAG_DONT_OVERWRITE_SETTINGS*-Flag gesetzt ist, da nach den Spezifikationen des Systemaufrufs bestehende Seitenrahmen nicht überschrieben werden drüfen.

10.4.5 Der Systemaufruf *map*

Der Systemaufruf *map* wird durch die Funktion *sysc_map* (*map.c*) implementiert. Zuerst prüft der Systemaufruf die verschiedenen Parameter und die Zugriffsrechte auf den Zielprozess. Anschließend lädt er verschiedene häufig verwendete Prozess-Eigenschaften und -Tabellen in lokale Variablen, darunter z.B. das Page Directory der Prozesse (*l__pdir_d* und *l__pdir_s*) oder die Speicherverwaltungstabelle des Zielprozesses (*l__pstat_d*).

Im zweiten Teil der Funktion werden dann in einer while-Schleife die Speicherseiten in den Zieladressraum eingeblendet. Dort werden zuerst die Seitentabellen des Ziel- und Quelladressraums ermittelt, die die zu ändernden Deskriptoren enthalten. Die Funktion blendet keine Speicherseiten in den Zieladressraum ein, wenn entweder dort im Zielbereich bereits Speicherseiten eingeblendet sind oder im Quelladressraum die ausgewählten Speicherseiten als inaktiv markiert sind. Ist dem Quellbereich nicht einmal eine Seitentabelle zugeordnet, werden 1024 Seiten übersprungen, um den Vorgang zu beschleunigen. Wenn jedoch eingeblendet werden kann, werden zu aller erst die PBT-Einträge angepasst. War eine Seite zuvor ohne PST-Liste, wird diese erstellt, andernfalls um die Daten des neuen Seitenmitbenutzers erweitert. Sind derzeit keine PST-Einträge mehr frei, wird die Operation mit einer Fehlermeldung abgebrochen.

Ist die Änderung des PBT-Eintrags erfolgreich gewesen, wird zuerst in der Speicherverwaltungstabelle des Zielprozesses der Eintrag frür die jeweilige Seitentabelle inkrementiert. Anschließend wird geprüft, ob es sich um ein Copy-On-Write-Mapping handelt. Ist dem so, wird ebenfalls der Eintrag des Quelladressraums für Copy-On-Write selektiert und der TLB für den Quelladressraum an der ge-

gebenen Adresse mit *INVLPG* (*mem.h*) invalidiert, da der Quelladressraum ja dem derzeitig aktiven Adressraum entspricht. Anschließend wird der Seitentabelleneintrag des Zieladressraum geändert. Ist der Zieladressraum mit dem Quelladressraum identisch - also somit auch derzeit aktiver virtueller Adressraum -, wird für diesen ebenfalls mit *INVLPG* der TLB für die gegebene Adresse invalidiert.

Zum Schluß werden Ziel und Quelladresse um 4096 erhöht und die Schleife wieder von vorne begonnen, es sei denn die angegebene Zahl von Seiten wurde abgearbeitet.

10.4.6 Der Systemaufruf *unmap*

Die Implementierung des Systemaufrufs *unmap* ist die Funktion *sysc_unmap* (*map.c*). Diese Funktion prüft ebenfalls zuerst alle Zugriffsrechte auf den Zielbereich und ermittelt Informationen, die während des eigentlichen Vorgangs später wichtig seien werden - darunter z.B. das Page Directory des Zielprozesses (*l__pdir_d*) und dessen Speicherverwaltungstabelle (*l__pstat_d*)

Im zweiten Teil der Funktion wird ebenfalls in einer while-Schleife die eigentliche Mapping-Operation durchgeführt. Zuerst wird die zuständige Seitentabelle mit *kmem_get_table* ermittelt. Hier ist anzumerken, dass *kmem_get_table* mitgeteilt wird, keine Seitentabelle zu erstellen, wenn bisher im Zieladdressraum keine existierte, da ja beim Unmap-Vorgang keine Seitenrahmen neu hinzukommen werden. Handelt es sich beim Zielbereich tatsächlich um einen Bereich, dem keine Seitentabelle zugeordnet wurde, werden ferner 1024 Seitenrahmen übersprungen, um den Vorgang zu beschleunigen. Je nachdem, welche Parameter *unmap* übergeben wurden, wird dann im folgenden Teil der Schleife die Zugriffsflags der virtuellen Seite geändert oder die virtuelle Seite komplett entfernt (*UNMAP_COMPLETE* oder *UNMAP_AVAILABLE*). In letzterem Fall erfolgt die Freigabe des zugeordneten Seitenrahmens (bzw. des *SID:VA*-Paares der zu entfernenden virtuellen Adresse aus der PST-Liste des PBT-Eintrags der Seite) über *kmem_free_user_page_frame* (*alloc.c*).

Wurde die Operation auf den aktuellen virtuellen Adressraum ausgeführt, wird mit *INVLPG* der TLB für die betroffenen Adressen invaildiert. Zum Schluß wird die Zieladresse um 4096 erhöht und die Schleife wiederholt, bis alle Seitenrahmen abgearbeitet wird.

10.4.7 Der Systemaufruf *move*

Der Systemaufruf *move* ist über die Funktion *sysc_move* (*map.c*) implementiert. Im wesentlichen ruft er intern die Implementierungen der Systemaufrufe *map* und *unmap* (*sysc_map* / *sysc_unmap*, *map.c*) auf. Tritt zwischen dem *map* und *unmap*-Vorgang ein Fehler auf, wird die Operation vorzeitig abgebrochen.

10.4.8 Der Systemaufruf *allow*

Auch wenn der Systemaufruf *allow*, der über die Funktion *sysc_allow* (*map.c*) implementiert ist, selbst keine Mapping-Operationen durchführt, ist sie der Vollständigkeit halber hier ebenfalls aufgeführt. Diese Operation ändert, nach Prüfung der Zugriffsrechte, lediglich die Thread-Deskriptoren, so dass die gewählten Zugriffsrechte für eine *map*- oder *unmap*-Operation hergestellt oder verweigert werden können.

10.4.9 Der Systemaufruf *io_alloc*

Eine Sonderrole spielt der Systemaufruf *io_alloc*, der zum Einblenden von Hardware-Speicherbereichen dient. Dessen Implementierung - *sysc_io_alloc* (*io.c*) - baut, neben den erforderlichen Prüfungen der Zugriffsrechte, im wesentlichen auf *kmem_map_pageframe_cur* auf, so dass eine weitere Beschreibung der internen Vorgänge nicht notwendig ist. Anzumerken ist, dass normalerweise das Flag *GENFLAG_NOT_CACHEABLE* gesetzt wird, um Zugriffe auf die Speicherseite unter Übergehung des

CPU-Caches möglich zu machen. Dies kann jedoch durch den Systemaufrufsparameter *IOMAP_WITH_CACHE* unterbunden werden.

10.5 Initialisierung der Speicherverwaltung

10.5.1 Initialisierung der Speicherverwaltungstabellen

Die Speicherverwaltung wird beim Start des Systems durch die Funktion *kmem_init_tables* (*meminit.c*) initialisiert. Diese Funktion berechnet zuerst die Startadressen und die Größe von IKP (wobei diese statisch ist), PBT, PST, NZS, HZS und Page-Buffer. Anschließend initialisiert sie PBT, PST, NZS und ggf. die HZS entsprechend.

In allen Einträgen der PBT wird der *usage*-Zähler, sowie die Einzelbenutzerdaten aus dem *single*-Element auf 0 gesetzt. In der PST werden alle Einträge zur Free-Liste verkettet, so dass alle Elemente der kompletten PST als freigegeben gelten. Der Anfang der Free-Liste der PST (*page_share_table_free, meminit.c*) wird auf den ersten Eintrag gelegt. Der letzte Eintrag der PST - und somit der Free-Liste erhält als folge Eintrag einen NULL-Zeiger.

In den NZS und den HZS werden jeweils die verbleibenden Speicherseiten des Page-Buffers der jeweiligen Zone als Einträge auf den Stack gelegt. Anschließend werden deren Stack-Pointer jeweils auf den zuletzt initialisierten Eintrag der beiden Stacks gelegt. Der HZS wird logischerweise nicht initialisiert, wenn es keine High-Zone gibt.

10.5.2 Initialisierung der Kernel-Seitentabellen

Nach diesen Initialisierungen wird die IKP initialisiert. Zuerst wird das Page-Directory (PD) der IKP auf die jeweiligen Page-Tables der IKP ausgerichtet - was nur die PD-Einträge der oberen 3 GiB betrifft. Zusätzlich wird der erste PD-Eintrag ebenfalls auf die erste Page-Table der IKP gestellt, da dies später für den Wechsel in den virtuellen Kerneladressraum wichtig ist.

Anschließend werden alle Kernel-Page-Tables mit Einträgen aufgefüllt, die die ersten 896 MiB des Adressraums in die ersten 896 MiB der oberen 3 GiB des virtuellen Adressraums abbilden. Die verbleiebenden Page-Tables werden mit Einträgen aufgefüllt, die auf keine Seitenrahmen verweisen. Diese werden später initialisiert. Bei den Seitentabelleneinträge, die das erste MiB des physikalischen Adressraums beschreiben, wird das Cache-Disable-Flag gesetzt, da dieser Speicherbereich auf PC-Plattformen sehr viele Hardware-Speicherseiten enthält, die nicht gecached werden dürfen (z.B. VGA-Textmodus-RAM). Für alle Speicherseiten und Seitentabellen des Kernels setzt das System das Flag *PFLAG_GLOBAL* (bzw. *GENFLAG_GLOBAL*). Dadurch soll, wenn von der jeweiligen Plattform unterstützt, der Zugriff auf den Kernel-Adressraum mittels der Paging-Global-Extension der x86-Architektur beschleunigt werden. Seiten mit gesetztem Global-Flag werden selbst bei einem Prozesswechsel nicht aus dem TLB entfernt. Werden sie geändert, muss immer der TLB durch die INVLPG-Instruktion invalidiert werden.

Ist die IKP initialisiert worden, versucht der Kernel in den neuen virtuellen Adressraum zu wechseln. Dies erledigt die Funktion *kmem_switch_krnl_spc* (*page.c*; *mem.h*). Diese Funktion führt als erstes den Plattformspezifischen Wechsel des Paging-Directories durch, indem das CR3-Register der CPU auf die Adresse des IKP-Pagedirectories gelegt wird. Anschließend wird das Paging-Bit im Kontrollregister CR0 gesetzt, wodurch das Paging in der CPU aktiviert wird. In diesem Moment sind Kernel-Code/Daten noch an der Stelle im virtuellen Adressraum eingeblendet, an der sie von GRUB geladen wurden. Gleichzeitig ist aber bereits der virtuelle Kerneladressraum jenseits der 3 GiB-Grenze aufgebaut. Um nun hier einen Wechsel zu vollziehen, werden die Daten- und Stacksegmentregister der CPU auf das Kernel-Datensegment der GDT gesetzt, dass seinen Beginn an der 3 GiB-Grenze hat. Anschließend wird ein Long-Jump in das gleichliegende Kernel-Code-Segment vollzogen, so dass nun der Kernel-Code ausgeführt wird, der im virtuellen Kerneladressraum jenseits

der 3 GiB-Grenze eingeblendet ist. Der Kernel-Code im alten Adressbereich (> 1 MiB) wird anschließend aus dem virtuellen Kerneladressraum ausgeblendet, so dass die IKP als Vorlage für andere Page-Directories verwendet werden kann.

Die Variable *i386_current_pdir* (*page.c*; *current.h*), welche angibt, welches Page-Directory derzeit verwendet wird, wird auf das Page-Directory der IKP gesetzt. Somit ist der virtuelle Kernel-Adressraum initialisiert und als gültiger virtueller Adressraum aktiviert.

10.6 Schlußbemerkungen

Die Speicherverwaltung ist einer der Hauptaufgaben des *hymk*. An einigen Stellen ist jedoch bereits jetzt Verbesserungsbedarf erkennbar. In diesem Abschnitt sind entsprechende Hinweise zur Speicherverwaltung aufgeführt.

10.6.1 Erweiterung um einen Paging-Dienst

Sollte eine Seitenauslagerung in HydrixOS auf Basis des *hymk* etabliert werden, so wird dringend eine Überprüfung des Codes empfohlen. In *kmem_get_table* (*page.h*) werden z.B. Seitenrahmen dann eingefügt, wenn der entsprechende Eintrag im Seitenverzeichnis kein gesetztes *GENFLAG_PRESENT*-Flag hat. Dementsprechend wäre es u.a. sinnvoll weitere Flags zu überprüfen, um zwischen nicht-presenenten (d.h. ausgelagerten) und inaktiven virtuellen Seiten zu unterscheiden. Ähnliches gilt für die Funktionen *map* und *unmap*.

Der Paging-Dämon müsste ebenfalls selbstständig Copy-On-Write-Vorgänge für nicht-presente Seitenrahmen durchführen. Er sollte es grundsätzlich vermeiden Nicht-Copy-On-Write-Shares auszulagern. Auch dies müsste nachträglich bedacht werden, ob und in wie fern Seitentabellen ausgelagert werden dürfen und welche Zugriffsrechte auf diese dem Paging-Dienst gewärleistet werden müssen und dürfen.

Ohne Änderung des Kernel-Codes ist derzeit eine Erweiterung um Seitenauslagerungsfähigkeiten des Gesamtsystems noch nicht möglich. Die Änderungen dürften aber nicht zu schwer sein. Der Aspekt wurde derzeit noch nicht beachtet, um später die Entwicklung eines Auslagerungsmechanismuses nicht durch ungünstige Vorgaben zu behindern.

10.6.2 Verbesserung der PST

Die PST ist derzeit ein statischer Puffer. Es sollte hier überlegt werden, wie die Einführung eines dynamischen Puffers sinnvoll und Speicherplatzschonend gestaltet werden kann.

10.6.3 DMA

Derzeit wurde die Verwendung eines DMA-Controllers noch nicht weiter beachtet. Da insbesondere die Intel-Architektur hier einige Restriktionen aufweist (u.a. müssen die Speicherseiten unterhalb der 1 MiB-Grenze liegen), müssten hier nachträglich besondere Allokationsmechanismen und evtl. ein zusätzlicher Systemaufruf für die Allokation von DMA-Speicher eingeführt werden - auch da ein DMA-Puffer normalerweise mehr als eine zusammenhängende physische Seite benötigt. Eine Mögilchkeit wäre, dass der Kernel für Treiber, die DMA-Puffer benötigen einige Kontingente an DMA-Puffern erst einmal bereithält und diese verbraucht, wenn Speicherknappheit eintritt. Die Sicherung der Kontingente für die jeweiligen Treiber würde ohnehin ja bereits zur Boot-Zeit des Systems geschehen oder könnte auch durch einen DMA-Dienst erfolgen.

Kapitel 11

Subjektenverwaltung

11.1 Die Infopages

Eine der Vorteile moderner Mikrokernel ist, dass das System gänzlich auf Systemaufrufe zur reinen Übergabe von Informationen vom Kernel an den Benutzer-Modus verzichtet. Statt dessen werden Informationen direkt über eingeblendete Speicherseiten ausgetauscht. Unter HydrixOS wurde hierbei versucht, den Aktualisierungsaufwand dieser Informationsseiten so niedrig wie möglich zu halten. Daher werden die Tabellen, die der Kernel für die Prozess- und Thread-Verwaltung verwendet direkt dem Benutzer-Modus zum Lesen zur Verfügung gestellt. Um aufwendiges Suchen in verketteten Listen zu vermeiden, werden diese in einem eigens dafür angelegten Speicherbereich des virtuellen Adressraums eingeblendet. Da der Kernel diese Bereiche ebenfalls verwendet, liegen sie im Bereich des Kernel-Adressraums jenseits der 3 GiB-Grenze.

11.1.1 Überblick über die Datenstrukturen

Alle Verwaltungstabellen im System sind als Arrays vom Typ *uint32_t* organisiert.

Die Haupt-Info-Page ist über den Zeiger *main_info* (*info.c*; *info.h*) erreichbar, wobei der Zeiger an die virtuelle Adresse der Hauptinfopage relativ zum Begin des Kernel-Adressraums (d.h. der Zeiger kann direkt aus dem Kernel-Code heraus verwendet werden) deutet. Deren physische Adresse kann über die Variable *real_main_info* ermittelt werden. Die einzelnen Elemente der Hauptinfopage sind jeweils die einzelnen Elemente des Arrays. Um die Übersicht des Programmtextes zu erhalten, existieren die *MAININFO_**-Makros, die gleichermaßen im Kernel-Code, wie auch im Code von Programmen für den Benutzer-Modus eingesetzt werden können, da die identische Header-Datei verwendet wird (*sysinfo.h*).

Die zum Anfang des Kernel-Adressraums relative virtuelle Adresse der Threadtabelle kann über den Zeiger *thread_tab* (*info.c*; *info.h*) referenziert werden. Der Array der Thread-Tabelle beschreibt tatsächlich alle Threads - d.h. jedes 8192. Element des Arrays entspricht einem weiteren Thread-Eintrag in der Thread-Tabelle, wobei auf einem x86-System natürlich nur die unteren 12-bit einer Thread-SID beachtet werden dürfen. Die Elemente jedes Thread-Eintrags können dann wiederum durch Addition über die *THRTAB_**-Makros (*sysinfo.h*) gleichermaßen in Kernel- und Benutzer-Code verwendet werden. Um den Quelltext weiter zu vereinfachen, wurde das *THREAD*-Makro (*info.h*) eingeführt, über das mittels Thread-SID und eines *THRTAB_**-Makros die jeweilige Information über einen Thread abgerufen oder geändert werden kann.

Für die Prozesstabelle gilt im wesentlichen das gleiche. Ihre relative virtuelle Adresse kann über den Zeiger *process_tab* (*info.c*; *info.h*) ermittelt werden. Auch in diesem Array beschreibt jeddes 8192. Element den nächsten Prozess. Die einzelenen Prozess-Informationen können durch Addition des jeweiligen *PRCTAB_**-Makros ermittelt werden. Auch hier existiert ein Makro zur Quelltextvereinfachung, das den Namen *PROCESS* (*info.h*) trägt und ebenfalls Thread-SID und das entsprechende

*PRCTAB_*-Makro zum Abruf oder zur Manipulation von Prozessinformationen als Parameter erhält.

Beide Tabellen bestehen - im Unterschied zur Hauptinfopage - nicht aus einem kontinuierlichen physischen Adressbereich. Die einzelnen Seitenrahmen eines Deskriptors werden durch Page-Mappings zusammengesetzt (der Mapping-Vorgang für Tabellendeskriptoren wird später erläutert). Alle Deskriptoren in einer der beiden Tabellen, die inaktiv sind, enthalten im ersten Element den Wert 0. Sie sind alle auf den gleichen leeren physischen Seitenrahmen, dessen physische Adresse über die Variable *real_empty_info* ermittelt werden kann.

11.1.2 Verwaltung von Subjektdeskriptoren

Die Verwaltung der Subjektdeskriptoren wird von zwei Funktionen übernommen: *kinfo_new_descr* (*info.c*; *info.h*) und *kinfo_del_descr* (*info.c*; *info.h*). Beide Funktionen erhalten als Parameter die SID des zu bearbeitenden Subjekts. Da die SID bereits die Information über die Subjektart enthält (Prozess oder Thread), entscheiden die Funktionen selbstständig, in welche der beiden Tabellen Deskriptoren einzublenden sind.

Die Funktion *kinfo_new_descr* liefert als Rückgabewert einen Zeiger auf den letzten verwendeten physikalischen Seitenrahmen des Deskriptors, da die physikalische Adresse des letzten Seitenrahmen bei der Threadverwaltung für dessen Thread-Local-Storage ständig verwendet wird (prinzipiell kann jedoch die Adresse jedes verwendeten physikalischen Seitenrahmens eines Deskriptors über den Seitendeskriptor des zugeordneten Prozess- oder Threadtabelleneintrags ermittelt werden).

Das Mapping in beiden Funktionen ist direkt implementiert. Zu beachten ist, dass beim Einblenden eines neuen Deskriptors in eine Tabelle neben *PFLAG_PRESENT* nur *PFLAG_USER* und *PFLAG_GLOBAL* gesetzt sein darf. Das Schreibzugriffsflag darf nicht gesetzt werden, da ansonsten Benutzerprogramme zugriff auf die Daten der Tabelle hätten (der Kernel kann auch bei gelöschtem Schreibzugriffsflag auf die Seite schreibend zugreifen). Bei jedem Mapping wird logischerweise der TLB mittels *INVLPG* invalidiert.

Der wesentliche Unterschied zwischen Installation eines neuen Deskriptors und der Entfernung eines bestehenden liegt darin, dass in einem Fall ein oder mehrere frisch alloziierte Seitenrahmen eingeblendet werden (*kinfo_new_descr*) und im anderen Fall *real_empty_info* an den Ort der Tabelleneinträge eingeblendet wird, um wieder einen leeren Eintrag zu erhalten (*kinfo_del_descr*). Wichtig ist noch, dass *kinfo_new_descr* jeweils bei der ersten Seite des neuen Deskriptors das erste *uint32_t*-Element jeweils auf den Wert 1 setzt, so dass der Deskriptor als aktiviert gilt.

11.1.3 Initialisierung des Infopage-Systems

Das Infopage-System wird durch die Funktion *kinfo_init* (*info.c*; *info.h*) initialisiert. Diese Funktion alloziiert physische Seitenrahmen für *real_main_info* und *real_empty_info* und blendet anschließend *real_main_info* im Bereich der Hauptinfopage als globale, für den benutzer nicht beschreibbare Speicherseite ein. Ebenfalls wird die Infopage auf einige initiale Informationswerte gesetzt, die Auskunft über die CPU, Speichergröße etc. geben. Hier ist zu beachten, dass bestimmte plattformspezifische Werte (z.B. CPU-Typ, Speichergröße etc.) deshalb schon vor dem Aufruf von *kinfo_init* initialisiert werden müssen.

Ferner initialisiert die Funktion die Prozess- und Thread-Tabelle, indem sie jede virtuelle Seite des Tabelle auf den physischen Seitenrahmen *real_empty_info* verweisen lässt (wobei der TLB natürlich ebenfalls invalidiert wird).

11.1.4 Die Makros der Hauptinfopage

Die für den Zugriff auf die Hauptinfopae verwendeten Makros haben derzeit folgende Bedeutung (siehe auch 8.1.1)

Makro	Bedeutung
MAININFO_CURRENT_PROCESS	SID des aktuellen Prozesses.
MAININFO_CURRENT_THREAD	SID des aktuellen Threads.
MAININFO_PROC_TABLE_ENTRY	Zeiger auf den aktuellen Prozesstabelleneintrag (absolute virtuelle Adresse).
MAININFO_THRD_TABLE_ENTRY	Zeiger auf den aktuellen Threadtabelleneintrag (absolute virtuelle Adresse).
MAININFO_KERNEL_MAJOR	Hauptversion des Kernels.
MAININFO_KERNEL_MINOR	Nebenversion des Kernels.
MAININFO_KERNEL_REVISION	Revisionsnummer des Kernels.
MAININFO_RTC_COUNTER_LOW	Zeit-Zähler des Kernels seit Systemstart (niederwertige 32-bit).
MAININFO_RTC_COUNTER_HIGH	Zeit-Zähler des Kernels seit Systemstart (höherwertige 32-bit).
MAININFO_CPU_ID_CODE	HydrixOS CPU-Identifikationsnummer.
MAININFO_PAGE_SIZE	Größe einer Speicherseite auf diesem System.
MAININFO_MAX_PAGE_OPERATION	Maximale Größe (in Seiten) einer Speicheroperation auf diesem System.
MAININFO_X86_CPU_NAME_PART_1	Erster Teil des CPU-Namens (x86-Spezifisch)
MAININFO_X86_CPU_NAME_PART_2	Zweiter Teil des CPU-Namens (x86-Spezifisch)
MAININFO_X86_CPU_NAME_PART_3	Dritter Teil des CPU-Namens (x86-Spezifisch)
MAININFO_X86_CPU_TYPE	CPU-Typ (x86-Spezifisch, siehe *CPUID*)
MAININFO_X86_CPU_FAMILY	CPU-Familie (x86-Spezifisch, siehe *CPUID*)
MAININFO_X86_CPU_MODEL	CPU-Modell (x86-Spezifisch, siehe *CPUID*)
MAININFO_X86_CPU_STEPPING	CPU-Stepping (x86-Spezifisch, siehe *CPUID*)
MAININFO_X86_RAM_SIZE	Größe des RAMs in Byte.

11.1.5 Die Makros der Prozesstabelle

Die für den Zugriff auf die Prozesstabelle verwendeten Makros haben derzeit folgende Bedeutung (siehe auch 8.1.2):

Makro	Bedeutung
PRCTAB_IS_USED	Gibt an, ob der Deskriptor derzeit verwendet wird oder nicht.
PRCTAB_IS_DEFUNCT	Gibt an, ob der Prozess durch *destroy_subject* inaktiviert wurde.
PRCTAB_SID	SID des Threads.
PRCTAB_CONTROLLER_THREAD_SID	SID des Kontroller-Threads des Threads.
PRCTAB_CONTROLLER_THREAD_DESCR	Zeiger auf den Deskriptor des Kontroller-Threads.
PRCTAB_PAGE_COUNT	Anzahl der vom Prozess verwendeten Speicherseiten.
PRCTAB_IO_ACCESS_RIGHTS	I/O-Zugriffsrechte des Threads (siehe *io_allow*)
PRCTAB_PAGEDIR_PHYSICAL_ADDR	Physikalische Adresse des Seitenverzeichnisses des Prozesses.
PRCTAB_IS_ROOT	Gibt an, ob der Prozess ein Root-Prozess ist (1) oder nicht (0).

Makro	Bedeutung
PRCTAB_IS_PAGED	Gibt an, ob der Prozess der Paging-Dämon ist (1) oder nicht (0).
PRCTAB_THREAD_COUNT	Anzahl der Threads des Prozesses
PRCTAB_THREAD_LIST_BEGIN	Erster Thread in der Thread-Liste des Prozesses.
PRCTAB_UNIQUE_ID	Eindeutige Prozesskennung (32-bit-Zahl)
PRCTAB_X86_MMTABLE	Anfang der im Deskriptor enthaltenen Speicherverwaltungstabelle des Prozesses

11.1.6 Die Makros der Threadtabelle

Die für den Zugriff auf die Threadtabelle verwendeten Makros haben derzeit folgende Bedeutung (siehe auch 8.1.3):

Makro	Bedeutung
THRTAB_IS_USED	Gibt an, ob der Deskriptor derzeit verwendet wird oder nicht.
THRTAB_SID	SID des Threads.
THRTAB_PROCESS_SID	SID des Dachprozesses.
THRTAB_SYNC_SID	SID der aktuellen *sync*-Operation.
THRTAB_MEMORY_OP_SID	Von *allow* freigegebene SID für Mapping-Operationen.
THRTAB_MEMORY_OP_DESTADR	Von *allow* freigegebene Zieladresse für Mapping-Operationen.
THRTAB_MEMORY_OP_MAXSIZE	Von *allow* angegebene Größe (in Seiten) des Zielbereichs für Mapping-Operationen.
THRTAB_MEMORY_OP_ALLOWED	Von *allow* freigegebene Mapping-Operationen.
THRTAB_KERNEL_STACK_ADDRESS	Adresse des Kernel-Stacks relativ zum Kernel-Adressraum.
THRTAB_PROCESS_DESCR	Direkter Zeiger zum Prozess-Deskriptor (relativ zum Kernel-Adressraum)
THRTAB_THRSTAT_FLAGS	Zustandsinformationen des Threads.
THRTAB_IRQ_RECV_NUMBER	Von *recv_irq* angefragter IRQ. 0xFFFFFFFF = *kein Thread*
THRTAB_FREEZE_COUNTER	Zähler, der angibt, wie viele *freeze_subject* Operationen auf diesen Thread angewandt wurden.
THRTAB_RUNQUEUE_PREV	Vorgänger des Threads in der Runqueue (*NULL*, wenn am Anfang oder - wenn *NEXT* gleich *NULL* - außerhalb der Runqueue).
THRTAB_RUNQUEUE_NEXT	Auf diesen Thread folgender Thread in der Runqueue (*NULL*, wenn am Ende oder - wenn PREV gleich *NULL* - außerhalb der Runqueue).
THRTAB_SOFTINT_LISTENER_SID	Thread der die Software-Interrupts dieses Threads mittels *recv_softint* derzeit abhört.
THRTAB_EFFECTIVE_PRIORITY	Effektive Priorität des Threads (plattformspezifischer Wert).
THRTAB_STATIC_PRIORITY	Statische Priorität des Threads.

Makro	Bedeutung
THRTAB_SCHEDULING_CLASS	Scheduling-Klasse des Threads
THRTAB_TIMEOUT_LOW	Zeitpunkt, an dem eine Timeout-behaftete Operation abgebrochen werden soll (niederwertige 32-Bit).
THRTAB_TIMEOUT_HIGH	Zeitpunkt, an dem eine Timeout-behaftete Operation abgebrochen werden soll (höherwertige 32-Bit).
THRTAB_LAST_EXCPT_NUMBER	Plattformunabhängige Nummer der letzten unbehandelten Exception, die in diesem Thread aufgetreten ist.
THRTAB_LAST_EXCPT_NR_PLATTFORM	Plattformspezifische Nummer der letzten unbehandelten Exception, die in diesem Thread aufgetreten ist.
THRTAB_LAST_EXCPT_ADDRESS	Adresse der letzten unbehandelten Exception, die in diesem Thread aufgetreten ist.
THRTAB_LAST_EXCPT_ERROR_CODE	Plattformspezifischer Fehler-Code der letzten unbehandelten Exception, die in diesem Thread aufgetreten ist.
THRTAB_PAGEFAULT_DESCRIPTOR	Page-Deskriptor des letzten unbehandelten Seitenfehlers, der in diesem Thread aufgetreten ist.
THRTAB_PAGEFAULT_LINEAR_ADDRESS	Lineare Adresse des letzten unbehandelten Seitenfehlers, der in diesem Thread aufgetreten ist.
THRTAB_PREV_THREAD_OF_PROC	Vorgehendes Mitglied der Liste der Threads des Dachprozesses (*NULL = Listenende*).
THRTAB_NEXT_THREAD_OF_PROC	Nachfolgendes Mitglied der Liste der Threads des Dachprozesses (*NULL = Listenende*).
THRTAB_RECEIVED_SOFTINT	Mittels *recv_softint* zuletzt empfangener Software-Interrupt.
THRTAB_RECV_LISTEN_TO	SID des Threads, dessen Software-Interrupts derzeit mittels *recv_softints* abgehört werden.
THRTAB_CUR_SYNC_QUEUE_PREV	Vorheriges Mitglied in der Sync-Warteschlange eines Threads, auf den derzeit mittels *sync* gewartet wird (*NULL* = Erstes Mitglied der Schleife).
THRTAB_CUR_SYNC_QUEUE_NEXT	Nächstes Mitglied in der Sync-Warteschlange eines Threads, auf den derzeit mittels *sync* gewartet wird (*NULL* = Letztes Mitglied der Schleife).
THRTAB_OWN_SYNC_QUEUE_BEGIN	Erstes Mitglied der eigenen Sync-Warteschlange.
THRTAB_TIMEOUT_QUEUE_PREV	Vorheriges Mitglied der Timeout-Warteschlange, falls Timeout gesetzt ist (*NULL* = erstes Mitglied).
THRTAB_TIMEOUT_QUEUE_NEXT	Nächstes Mitglied der Timeout-Warteschlange, falls Timeout gesetzt ist (*NULL* = letztes Mitglied).
THRTAB_UNIQUE_ID	Eindeutige Threadkennung (32-bit-Zahl)
THRTAB_X86_KERNEL_POINTER	Stack-Pointer des Kernel-Stacks

Makro	Bedeutung
THRTAB_X86_TLS_PHYS_ADDRESS	Physikalische Adresse des Seitenrahmens, der den *Thread local storage* enthält.
THRTAB_X86_FPU_STACK	Anfang des 512 Byte großen Bereichs im Deskriptor, in den der aktuelle FPU / MMX (etc.)-Zustand gespeichert werden soll.
THRTAB_LOCAL_STORAGE_BEGIN	Anfang des *Thread local storage*.

11.2 Threadverwaltung

11.2.1 Kernel-Stacks

Aufbau eines Kernel-Stacks

Eine weitere wichtige Datenstruktur des Kernels sind die sog. Kernel-Stacks. Jeder Thread besitzt neben seinem normalen User-Mode-Stack, auf den der Benutzer-Code des Threads ohne weiteres zugreifen kann, einen Kernel-Mode-Stack, der vor allen Zugriffen aus dem Benutzermodus geschützt ist.

Immer dann, wenn ein Thread in den Kernel-Modus wechselt, wird mittels einer speziellen Fähigkeit der x86-Architektur der Stackpointer auf einen vom Kernel festgelegten Kernel-Mode-Stack gewechselt, um anschließend darauf den aktuellen Benutzermodus-Zustand (Registerinhalte, Programmzeiger etc.) des unterbrochenen Threads zu speichern. Ferner wird der Kernel-Stack innerhalb des Kernel-Codes als regulärer Stack verwendet.

Ein Kernel-Stack ist genau eine Speicherseite (also 4 KiB) groß. Die letzten 68 Byte[1] enthalten (derzeit) stets den aktuellen CPU-Zustand (abgesehen des FPU-Stack / der XMM-Register etc.). Wobei der Aufbau dieser 68 Byte derzeit wie folgt ist:

DWord ab Blockbeginn	**Register**	Anmerkungen
0	EDI	
1	ESI	
2	EBP	
3	(Ungenutzt)	*Siehe x86-Handbuch: PUSHA*
4	EBX	
5	EDX	
6	ECX	
7	EAX	
8	GS	GS-Datensegment im Benutzer-Modus (*DS_USER*)
9	FS	FS-Datensegment im Benutzer-Modus (*DS_USER*)
10	ES	ES-Datensegment im Benutzer-Modus (*DS_USER*)
11	DS	DS-Datensegment im Benutzer-Modus (*DS_USER*)

[1] Dies ergibt sich aus der Tatsache, dass es sich um einen Stack handelt, der nach dem LIFO-Prinzip arbeitet. Wenn ein Prozess aus dem Benutzer-Modus in den Kernel-Modus wechselt, ist der zugehörige Stackpointer auf die oberste Adresse des Stacks gerichtet. Beim Ablegen von Daten wird der Stackpointer dekrementiert und die Daten in abwärtiger Richtung auf dem Stack gespeichert.

DWord ab Blockbeginn	Register	Anmerkungen
12	EIP	Programm-Zeiger im Benutzer-Modus (*DS_USER*)
13	CS	Codesegment im Benutzer-Modus (*CS_USER*)
14	EFLAGS	Flag-Register
15	ESP	Stack-Zeiger des Benutzer-Stacks
16	SS	Stacksegment im Benutzer-Modus (*DS_USER*)

Verwaltung von Kernel-Stacks

Zur Einrichtung und Vernichtung von neuen Kernel-Stacks existieren die Funktionen *ksched_new_stack* (*frame.c*; *sched.h*) und *ksched_del_stack* (*frame.c*; *sched.h*). Diese Funktionen rufen derzeit lediglich die normalen Speicherverwaltungsroutinen zur Allokation bzw. Freigabe einer Kernel-Seite auf.

Neben diesen reinen Verwaltungsfunktion existiert noch eine Funktion zur Initialisierung eines Kernel-Stacks *ksched_init_stack* (*frame.c*; *sched.h*). Diese Funktion initialisiert einen neuen Kernel-Stack. Sie setzt dabei den Code-Zeiger für den Betrieb im Benutzermodus des neuen Threads an dessen gewünschte Anfangsadresse. setzt im EFLAGS-Register das Interrupt-Flag, so dass mit dem neuen Thread die IRQs aktiv werden. Den Code- und Daten-Segmenten werden die verschiedenen Standard-Segmente zugewiesen. Als Rückgabewert liefert die Funktion einen Stack-Zeiger. Die Funktion kann wahlweise auch Stacks für Threads erstellen, die gleich von Anfang an im Kernel operieren und nicht sofort in den Benutzermodus wechseln.

Dazu muss der Parameter *mode* der Funktion auf *KSCHED_KERNEL_MODE* gesetzt werden. Hierbei werden die jeweiligen Segmentdeskriptoren auf Kernel-Segmente gesetzt. Außerdem wird kein Stacksegment für den Kernel eingerichtet, da die *IRET*-Instruktion, die zur Rückkehr von Interrupts eingesetzt wird, bei einer Rückkehr eines Programms in den gleichen Betriebsmodus (Kernel-Mode nach Kernel-Mode) keine Angaben über den Stack erwartet.

Soll eine Rückkehr in den Benutzermodus beim Ersten Start des Threads erreicht werden, ist der Parameter *mode* auf *KSCHED_USER_MODE* zu setzen. Derzeit wird *KSCHED_KERNEL_MODE* nur bei der Erzeugung des *idle*-Threads verwendet, da dieser im Kernel-Code enthalten ist (dies wird später noch einmal genauer erläutert).

11.2.2 Erzeugung von Threads

Für die Erzeugung neuer Threads ist die interne Funktion *ksubj_create_thread* (*subject.c*) zuständig. Diese Funktion wird von den Systemaufrufen zur Thread- und Prozesserzeugung verwendet und konfiguriert einen neuen Thread nur so, dass er für beide Arten von Threaderzeugung weiterverwendet werden kann.

Die Suche nach einer freien Thread-SID erfolgt derzeit einfach dadurch, dass die Thread-Tabelle nach leeren Deskriptoren durchsucht wird. Wurde ein inaktiver Deskriptor gefunden, wird die SID des Deskriptors errechnet und über *kinfo_new_descr* (*info.c*; *info.h*) ein neuer Deskriptor erzeugt. Der Rückgabewert von *kinfo_new_descr* gibt, wie bereits erwähnt, die physikalische Adresse des letzten Seitenrahmens zurück. Da dieser Seitenrahmen für den *Thread local storage* des neuen Threads verwendet werden soll, wird diese Adresse im *THRTAB_X86_TLS_PHYS_ADDR*-Feld des Deskriptors gespeichert. Anschließend wird ein neuer Kernel-Stack mittels *ksched_new_stack* erstellt und im weiteren Verlauf durch *ksched_init_stack* (*frame.c; sched.h*) mit *KSCHED_USER_MODE* als Benutzer-Modus-Thread initialisiert.

Der neue Thread liegt nach dem Aufruf noch nicht in der Ausführungswarteschlange des Schedulers vor und gehört dem aktuellen Prozess. Er ist allerdings noch nicht in die Liste der Threads

des Prozesses eingereiht worden. Nun ist es Aufgabe der verschiedenen Systemaufrufe, den Thread weiter zu initialisieren.

11.2.3 Der Systemaufruf *create_thread*

Die Implementierung des Systemaufrufs *create_thread* - die Funktion *sysc_create_thread* (*subject.c*; *sysc.h*) baut im wesentlichen auf *ksubj_create_thraed* (*subject.c*) auf und führt nach Erstellung des Threads durch diese nur noch weitere Schritte zur Initialisierung des Threads als weiteren Thread des aktuellen Prozesses durch. So wird u.a. der neue Thread in die Thread-Liste des aktuellen Prozesses eingereiht.

11.2.4 Vernichtung von Threads

Da innerhalb des Kernels keine interne Verwendung für die Vernichtung von Threads besteht, ist die dafür zuständige Routine direkt in den Systemaufruf *sysc_destroy_subject* (*subject.c*; *sysc.h*) implementiert. Da diese Funktion auch für die Blockade von Prozessen zuständig ist, liegt der Abschnitt zur Vernichtung von Thread-Subjects in folgendem If-Block:

```
if ((sid & SIDTYPE_THREAD) == SIDTYPE_THREAD)
{
    ...
}
```

Nachdem die verschiedenen Ausführungsvoraussetzungen geprüft wurden (Gültiger Thread; Thread ist nicht aktueller Thread;), wird der Thread aus der Run-Queue und der Thread-Liste seines Prozesses entfernt. Anschließend werden Threads aufgeweckt, die versucht haben, eine *sync*-Operation auszuführen. Hat der Thread selbst derzeit eine *sync*-Operation durchgeführt, so wird er aus der Liste des Zielthreads der *sync*-Operation entfernt[2]. Ist der Thread der letzte Thread des Prozesses wird er im Anschluß durch Aufruf der Funktion *ksubj_kill_proc* (*subject.c*) ebenfalls beendet. Sein Kernel-Stack wird durch *ksched_del_stack* (*frame.c*; *sched.h*) freigegeben. Hat der Thread zuvor auf eingehende IRQs gewartet, so wird der IRQ mittels *kio_renable_irq* (*io.c*; *sched.h*) wieder freigegeben. Threads, die Software Interrupts des aktuellen Threads abgreifen, werden daraufhin ebenfalls gestartet. Gleichermaßen wird der Zugriff auf Threads, deren Software Interrupts durch den zu zerstörenden Thread abgehört werden, aufgehoben. Schließlich wird der Thread-Deskriptor durch *kinfo_del_descr* (*info.c*; *info.h*) freigegeben.

11.3 Prozessverwaltung

11.3.1 Erzeugen von Prozessen

Der Kernel hat keinen Bedarf für eine interne Funktion zur Erzeugung von Prozessen. Die Erzeugung von Prozessen wird daher vollständig von der Behandlungsroutine des Systemaufruf *create_process*, *sysc_create_process* (*subject.c*, *sysc.h*) übernommen.

Als erstes wird von dieser Funktion ein neuer Deskriptor ermittelt, indem einfach in einer Schleife nach dem nächsten freien Deskriptor gesucht wird. Der neue virtuelle Adressraum des Prozesses

[2] Zu den verschiedenen Operationen, bei denen Threads sich mit dem zu zerstörenden Threads synchronisieren oder dieser sich mit anderen synchronisiert (*sync*, *recv_softints*), wird zum besseren Verständnis die Dokumentation der jeweiligen Mechanismen empfohlen (u.a. für die Funktionen *ksync_interrupt_other* und *ksync_remove_from_waitqueue*, so wie die verschiedenen Timeout-Operationen).

wird über *kmem_create_space* (*page.c, mem.h,* siehe 10.4.1) erzeugt. Der neue Deskriptor wird über *kinfo_new_descr* (*info.c, info.h*) erzeugt.

Gemäß den Spezifikationen wird ein initaler Thread erzeugt. Dies geschiet über *ksubj_create_thread* (*subject.c*). Der neue Thread erlaubt dem Prozess, der den Aufruf *create_process* gestartet hat, Speicherseiten einzublenden. Der neue Thread ist zudem nach dem Start vorerst noch eingefrohren und muss mit *awake_subject* reaktiviert werden. Der neue Thread wird als Controller-Thread des neuen Prozesses festgelegt.

Ist der Erzeguerprozess ein Root-Prozess, so werden die Root-Rechte weitervererbt.

11.3.2 Stoppen von Prozessen

Ein Prozess kann über den Aufruf *destroy_subject* durch den Paging-Dämon blockiert werden, so dass keiner seiner Threads wieder aktiv werden kann, um anschließend jeden Thread des Prozesses einzeln mit *destroy_subject* beenden zu können. Die Implementierung von *destroy_subject* ist in *sysc_destroy_subject* (*subject.c, sysc.h*) enthalten. Da diese Funktion ebenfalls für die Vernichtung von Threads zuständig ist, ist der für Prozesse zuständige Teil in folgendem If-Block enthalten:

```
if ((sid & SIDTYPE_PROCESS) == SIDTYPE_PROCESS)
{

    ...

}
```

In diesem Block wird im wesentlichen der Prozess durch das Flag *PRCTAB_IS_DEFUNCT* als suspendiert gemeldet. Ebenfalls werden alle Threads des Prozesses aus der Runqueue entfernt.

11.3.3 Vernichten von Prozessen

Ein Prozess wird immer dann vernichtet, wenn sein letzter Thread vernichtet wird. Dies geschiet in der Funktion *sysc_destroy_subject* (*subject.c, sysc.h*) in folgendem If-Block, der dem Routinenteil zur Threadvernichtung untergeordnet ist 11.2.4:

```
/* Decrement the thread counter of the process */
l__process[PRCTAB_THREAD_COUNT] --;

/* Last thread? Kill the process, too. */
if (l__process[PRCTAB_THREAD_COUNT] == 0)
{

    ksubj_kill_proc(l__process);

}
```

Der Aufruf *ksubj_kill_proc* (*subject.c*) ist also für die Vernichtung von Prozessen zuständig und wird gestartet, wenn der Thread-Counter des Prozesses 0 erreicht hat. Der Unteraufruf gibt im wesentlichen anhand der Speicherverwaltungstabelle des Prozesses die Seitentabellen des Prozesses wieder frei und zerstört den Adressraum über *kmem_destroy_space* (*page.c, mem.h,* siehe 10.4.1). Ebenfalls wird der Deskriptor über *kinfo_del_descr* (*info.c, info.h*) freigegeben.

11.3.4 Ändern des Controller-Threads

Der Controller-Thread eines Prozesses wird durch den Systemaufruf *set_controller*, implementiert durch *sysc_set_controller* (*subject.c*, *sysc.h*), geändert. Der Aufruf prüft lediglich ob der Aufrufer die nötigen Zugriffsrechte auf den Zielthread hat und ändert anschließend den Prozesstabellen-Eintrag *PRCTAB_CONTROLLER_THREAD_SID* und *PRCTAB_CONTROLLER_THREAD_DESCR* des Prozessdeskriptors entsprechend.

11.3.5 Root-Mitgliedschaft ändern

Der Systemaufruf zum Vergeben oder Aufgeben der Root-Rechte *chg_root* wird durch die Routine *sysc_chg_root* (*security.c*, *sys.h*) implementiert. Zu diesem Systemaufruf ist nichts weiteres zu erwähnen. Er überprüft die Zugriffsrechte, ändern anschließend den Eintrag *PRCTAB_IS_ROOT* entsprechend der Aufrufparameter und passt die IO-Zugriffsrechte des Prozesses entsprechend an.

Kapitel 12

Die Ausführungsschicht

12.1 Kernel-Eintritt

Immer wenn ein Thread einen Systemaufruf tätigt, durch einen IRQ unterbrochen wird, einen ungültigen Software-Interrupt auslöst oder eine Exception verursacht, findet ein Wechsel vom Benutzer-Modus in den Kernel-Modus statt. Für jede dieser vier Möglichkeiten existieren im Kernel unterschliedliche Kernel-Eintrittsverfahren, die nun in diesem Abschnitt näher beleuchtet werden, wobei in diesem Abschnitt nur der Einsprung selber erläutert wird, nicht die Aktion, für die der Thread eigentlich in den Kernel gewechselt hat.

Die Kernel-Einsprungspunkte für IRQs, Exceptions und ungültige Software-Interrupts sind in der Datei *irq.s* definiert. Die Kernel-Einsprungspunkte für Systemaufrufe in der Datei *sysc.s*, da für jeden Systemaufruf ein eigener Einsprungscode verwendet wird.

12.1.1 Die Zugriffsmodi der x86-Architektur

Die x86-Architektur verfügt im Protected Mode über vier verschiedene Betriebsmodi, die auch als „Ring“ bezeichnet werden. Davon sind die ersten drei Betriebsmodi (Ring 0 bis 2) für das Betriebssystem und der letzte Betriebsmodus (Ring 3) für Anwendungsprogramme gedacht. Je nach Betriebsmodus der CPU sind verschiedene Zugriffe und Instruktionen erlaubt oder verboten, wobei im Ring-0-Modus maximale Zugriffsrechte bestehen. Der *hymk* verwendet derzeit nur Ring 0 (Kernel-Modus) und Ring 3 (Benutzer-Modus).

12.1.2 Die Interrupt Descriptor Table

Soll von einem Modus zum anderen gewechselt werden, so ist dies nur über Far-Jumps, Interrupts und sog. Call-Gate-Deskriptoren möglich, wobei der *hymk* nur Wechsel über Interrupts (IRQs, Software-Interrupts/Systemaufrufe, Exceptions) gestattet. Für jeden Interrupt ist hierbei in der sog. Interrupt Deskriptor Table - die IDT - (definiert durch *i386_idt_s* in *start.s*) ein sog. Interrupt-Gate-Deskriptor (siehe *idt_t* in *sched.h*) abgelegt.

In dieser Tabelle sind gemäß den IA-32-Spezifikationen die ersten 32 Interrupts für die Exceptions belegt. Der *hymk* belegt außerdem für die IRQs die Interrupts *0xA0 - 0xAF* und für die Systemaufrufe die Interrupts *0xC0 - 0xD5* (wobei in späteren hymk-Versionen weitere Interrupts belegt werden könnten). Alle anderen Software-Interrupts werden nicht verwendet. Ihr Aufruf führt zu keiner Aktion. Der Aufruf von Exception- und IRQ-Handlern führt zu einer Schutzverletzung (es sei denn *recv_softints* wird auf den Thread angewandt), wobei die Breakpoint-Exception, die Overflow-Exception und die Bound-Range-Exceed-Exception (*0x3 - 0x5*) auch vom Benutzermodus aufgerufen werden können.

Soll also in den Kernel-Modus gewechselt werden, ist ein Interrupt erforderlich. Ein Software-Interrupt wird über die Instruktion *INT* ausgelöst, eine Exception durch einen Fehler/Trap oder die

Instruktion *INTO* (Overflow-Exception), einen IRQ durch ein externes Gerät. In jedem Fall lädt der Prozessor den Kernel-Stack und dessen Stacksegment anhand zuvor festgelegter Daten (wie später erläutert wird). Auf den Kernel-Stack werden dann das im Benutzermodus verwendete Stack-Segment (*SS*) und der zugehörige Stack-Pointer (*ESP*), die Maschienenflags (*EFLAGS*), das Code-Segment (*CS*) und der Programmzeiger (*EIP*) durch die CPU gespeichert. Anschließend wird die in der IDT festgelegte Routine aufgerufen, wobei auf Grund des IDT-Eintrags als Code-Segment das Kernel-Codesegment verwendet wird.

12.1.3 Kernel-Einsprung durch IRQs

Für alle IRQs lautet diese Low-Level-Routine *i386_irqhandleasm_0 - i386_irqhandleasm_15* (*irq.s, sched.h*). Damit diese Routinen nicht tatsächlich 16 Mal implementiert werden müssen, bedient sich der *hymk*-Code Assembler-Makros. Das für die IRQ-Routinen zuständige Makro heißt *MIRQ*. Als Parameter erhält es einerseits die Nummer des IRQs. Andererseits erhält es auch das Kontroll-Port des zu verwendenden Interrupt-Controlers, da alle IRQs ab dem IRQ 8 einen anderen Controler verwenden. Die Portnummern sind *0x20* für den Master-Controller und *0xA0* für den Slave-Controller.

Die Behandlungsroutine selbst sperrt alle IRQs durch das Löschen des Interrupt-Flags und sichert zuerst die restlichen Segmentregister (*DS, ES, FS, GS)*, dann die übrigen generellen Register (siehe auch 11.2.1). Anschließend werden die Datensegmentregister des Kernels geladen. Die darauf folgenden Codeabschnitte sind nur noch für die IRQ-Behandlung interessant: dort wird die IRQ-Nummer auf den Stack gelegt, die Kernel-Behandlung gestartet und der IRQ als abgearbeitet markiert - die genauen Abläufe werden in späteren Abschnitten näher erläutert.

12.1.4 Kernel-Einsprung durch Exceptions

Die Behandlung von Exceptions läuft im wesentlichen ähnlich ab. Die Routinen zur Exception-Behandlung heißen *i386_exhandleasm_0 - i386_exhandleasm_31* (*irq.s, sched.h*). Auch sie sind über ein Assembler-Makro realisiert. Dieses Makro trägt den Namen *MEX*. Als Parameter erhält das Makro einerseits die Exception-Nummer und andererseits den Hinweis darauf, ob die Exception mit einem Fehlercode verbunden ist oder nicht. Die Routine unterscheidet sich zur IRQ-Routine nur am Anfang. Dort wird zuerst der Fehlercode des Systems in verschiedene Puffer gesichert (dazu später mehr), bevor der Prozessorstatus auf dem Stack gespeichert wird. Existiert ein Fehlercode, so wird dieser zuvorls gesichert und vom Stack geholt. Anschließend wird wie bei einer IRQ-Routine verfahren und darauf die High-Level-Exceptionbehandlung gestartet.

12.1.5 Kernel-Einsprung durch Systemaufrufe

Auch Systemaufrufe werden ähnlich eingeleitet, allerdings besteht hier für jeden Systemaufruf eine eigene Implementierung, die jeweils den Namen des Systemaufrufs mit dem Prefix *i386_sysc_* trägt (siehe *start.s, sysc.h*). Hier besteht der Unterschied darin, dass der Software-Interrupt an die Routine *kremote_received* weitergegeben werden kann, wenn die Umleitung von Software-Interrupts für diesen Thread aktiviert ist - bei IRQs und Exceptions geschiet dies in einem anderen Codeteil (dies wird später näher erläutert). Auch hier wird der CPU-Status auf den Stack geschrieben und anschließend die High-Level-Routine (*sysc_**) gestartet.

12.1.6 Kernel-Einsprung durch unbelegte Software-Interrupts

Schließlich gibt es noch 187 Software-Interrupts, die weder einem IRQ, noch einer Exception, noch einem Systemaufruf zugewiesen sind. Diese Software-Interrupts können prinzipiell aus dem Benutzermodus aufgerufen werden, führen aber zu keiner Aktion - es sei denn, auf den Thread wurde eine

Umleitung der Software-Interrupts mittels *recv_softints* angewendet. Diese Interrupts dürfen nicht gesperrt sein, da ja u.a. bei der Emulation fremder Betriebssysteme deren Systemaufrufe nicht über die gleichen Software-Interrupts wie die des *hymk* laufen.

Die Low-Level-Routine für unbelegte Softwareinterrupts trägt den Namen *i386_emptyint_handler* (*irq.s, sched.h*). Im Unterschied zu den übrigen Einsprungroutinen, ist dieser Routine nicht direkt bekannt, welcher Interrupt Auslöser des Aufrufs war. Diese Routine muss - nach der üblichen Sicherung des Prozessorstatus - dies selbst herausfinden. Da als Quelle nur Software-Interrupts über den *INT*-Befehl in Frage kommen, liest die Funktion einfach an der zuletzt ausgeführten Adresse im Benutzer-Modus den ein Byte großen Parameter des *INT*-Befehls nach. Die High-Level-Behandlung (dabei u.a. die Weiterleitung an andere Threads, falls *recv_softints* angewendet wurde) erfolgt über die Routine *i386_handle_emptyint* (*intr.c*, *sched.h*).

12.2 Kernel-Austritt

Ist ein Systemaufruf ausgeführt, ein IRQ, eine Exception oder ein unbelegter Software-Interrupt behandelt worden, so findet wieder ein Rückwechsel in den Benutzer-Modus statt. Jenachdem ob und wie der Scheduler zuvor entschieden hat, wird dabei ebenfalls ein Threadwechsel durchgeführt. In der Regel wird der Kernel-Austritt dadurch eingeleitet, dass die High-Level-Behandlungsroutine durch *return* wieder zur Low-Level-Routine zurückkehrt. Diese führt ggf. noch weitere spezifische Anweisungen aus (z.B. Speichern des Rückgabewerts eines Systemaufrufs). Zur eigentlichen Rückkehr aus dem Systemaufruf kommt es dann durch einen Sprung in den Codeabschnitt *i386_do_context_switch* (*irq.s*)[1].

12.2.1 Weitere Wege zum Kernel-Austritt

Ein weiterer Weg zur Rückkehr aus dem Kernel- in den Benutzer-Modus ist die Aufgabe der CPU-Zeit eines Threads während der Ausführung im Kernel über den Aufruf *i386_yield_kernel_thread* (*irq.s, current.h*), bzw. über das darauf aufbauende Makro *SYSCALL_YIELDS_THREAD*. Dies ist z.B. erforderlich, wenn ein Thread während einer *sync*-Operation auf die Gegenseite warten muss. Diese Yield-Operation sichert den aktuellen Kernel-Zustand auf dem Stack und speichert als Rückkehradresse bei erneuter Aktivierung des Threads die Funktion *i386_awake_kernel_thread* (*irq.s*), die durch ein einfaches Return die Rückkehr zur unterbrochenen Kernel-Routine einleitet.

12.2.2 Die Austrittsroutine

Der eigentliche Austritt aus dem Kernel-Modus findet also über die Routine *i386_do_context_switch* (*irq.s*) statt. Wie der Name schon sagt, kann diese Routine zu einem Kontext-Wechsel führen (sie muss aber nicht). Die Routine kümmert sich effektiv nur um den Wechsel zwischen Kernel- und Benutzermodus und ggf. um den Wechsel des aktuellen Threads. Die Auswahl des neuen Threads und der Wechsel des virtuellen Adressraums wird von der Scheduler-Funktion *ksched_next_thread* (*current.c, sched.h*) übernommen, die in späteren Abschnitten näher erläutert wird.

Argumente des Aufrufs

Da es sich nicht um einen Funktionsaufruf im eigentlichen Sinne handelt, werden die Argumente dieser Routine nicht über den Stack, sondern über feste Puffer übergeben[2]. Diese Puffer sind:

[1] Hier sei angemerkt, dass es sich wirklich um einen Codeabschnitt und nicht um eine Unterroutine handelt, da werder Rücksprungadresse noch etwaige Parameter auf dem Stack gespeichert werden.

[2] bei einer SMP-Implementierung des *hymk* muss dies geändert werden.

ksched_change_thread Diese Variable gibt an, ob ein Kontext-Wechsel stattfinden soll (1) oder nicht (0).

i386_old_stack_pointer Diese Variable ist ein Zeiger auf einen Puffer, in den der Stackpointer des Kernel-Stacks des aktuellen Threads gespeichert werden soll, wenn ein Thread-Wechsel stattfindet. Dieser Puffer selbst ist das Element *THRTAB_X86_KERNEL_POINTER* im Deskriptor des alten, auszutauschenden Threads.

i386_new_stack_pointer Diese Variable ist ein Zeiger auf einen Puffer, der den Stackpointer des Kernel-Stacks des Threads enthält, der aktiviert werden soll, wenn ein Thread-Wechsel stattfindet. Dieser Puffer selbst ist das Element *THRTAB_X86_KERNEL_POINTER* im Deskriptor des neu zu ladenden Threads.

kinfo_io_map Diese Variable ist ein Zeiger auf die IO-Zugriffsrechte des zu ladenden Threads (dies ist für das Setzen der Zugriffsrechte auf die I/O-Ports wichtig). Die IO-Zugriffsrechte stehen im Element *PRCTAB_IO_ACCESS_RIGHTS* im Prozess-Deskriptor des neu zu ladenden Threads.

i386_esp0_ret Diese Variable ist ein Zeiger auf das ESP0-Feld des Kernel-TSS. Der Zeiger selbst ist während des Systembetriebs konstant. Beim Wechsel aus dem Benutzer-Modus in den Kernel-Modus setzt die CPU den Stackpointer auf den Wert des ESP0-Feldes. Das ESP0-Feld wird von *i386_do_context_switch* auf die Kernel-Stackadresse gesetzt, bei der der Kernel in den Benutzermodus gewechselt hat.

Ablauf des Austritts

Als erstes wird von *i386_do_context_switch* über den Wert der Variable *ksched_change_thread* (*schedule.c, sched.h*) geprüft, ob ein Threadwechsel stattfinden soll. Soll kein Wechsel stattfinden, wird direkt zur eigentlichen Rückkehrroutine *i386_do_context_switch_ret* gewechselt. Andernfalls wird mit dem Threadwechsel begonnen. Zuerst wird *ksched_change_thread* wieder auf 0 zurückgesetzt, so dass die Variable nach Abschluß der Routine keinen undefinierten Wert hat. Anschließend wird der Kernel-Stackzeiger des auszutauschenden Thread über den Zeiger *i386_old_stack_pointer* (*irq.s, sched.h*) in dessen Deskriptor gespeichert, so dass bei späterem Aufruf des Threads wieder der korrekte Kernel-Stack geladen werden kann. Aus dem Puffer, auf den *i386_new_stack_pointer* (*irq.s, sched.h*) zeigt, wird anschließend der Kernel-Stackzeiger des neuen Threads geladen. Anschließend ist prinzipiell der für den Threadwechsel spezifische Teil getan.

Nun wird der Rückkehrzeiger für den Kernel-Stack im TSS-Segment (*ESP0*) über den Zeiger *i386_esp0_ret* (*irq.s, sched.h*) auf den Wert des neuen Stackpointers gesetzt. Wenn ein Interrupt auftritt, wird die CPU genau diesen Zeiger als Stackzeiger des Kernel-Modus-Stacks verwenden.

Ist im IO-Zugriffsfeld des neuen Prozesses das Flag *IO_ALLOW_PORTS* gesetzt, was anhand von *kinfo_io_map* (*info.c, info.h*) geprüft wird, wird das sog. IOPL (I/O priviledge level) der CPU auf 3 gesetzt, wodurch der nun startende Ring-3-Code Zugriff auf die I/O-Ports erhält. Andernfalls wird das IOPL auf 0 gesetzt und somit der Zugriff auf die Ports gesperrt.

Schließlich folgt der Abschnitt *i386_do_context_switch_lastret* der nun die eigentliche Rückkehr ausführt. Es werden zuerst die auf dem Kernel-Stack beim Kernel-Eintritt gespeicherten Register vom Stack geladen und anschließend durch ein *IRET* von der CPU eine Rückkehr in den Benutzermodus anhand der auf dem Kernel-Stack gespeicherten Daten über den Benutzermodus-Wert des Stacksegments (*SS*), des Stackzeigers (*ESP*), des Flag-Registers (*EFLAGS*), des Code-Segment (*CS*) und des Programmzeigers (*EIP*) durchgeführt. Da im EFLAGS-Register im CPL-Feld der Wert 3 steht, weiß die CPU, dass ab jetzt wieder die unprivilegierte Ausführung stattfinden soll.

12.3 Zusammenfassung des Ein- und Austrittsmechanismus

Um den Mechanismus nun noch einmal jenseits der Implementationsdetails zu betrachten: Im wesentlich wird beim Kernel-Eintritt auf dem Kernel-Stack der Zustand der CPU im Benutzermodus gespeichert, so dass bei Abruf dieser Daten die Ausführung im Benutzer-Modus fortzusetzen. Beim Kernel-Austritt werden diese Daten wiederhergestellt[3]. Bei einem Threadwechsel wird lediglich der Stack vor der eigentlichen Rückkehr gewechselt, so dass bei der Rückkehr nun die Zustandsdaten eines anderen, zuvor unterbrochenen Threads geladen werden. Grundsätzlich kann *i386_do_context_switch* nicht aufgerufen werden, sondern steht immer am Schluß einer Eintrittsroutine, da ja der Zeiger des Kernel-Stacks wieder an der Stelle angelangt sein muss, an der beim Eintritt die Register gespeichert wurden.

Eine Ausnahme hierbei bildet *SYSCALL_YIELDS_THREAD*. Hier wird vor Beendigung einer Kernel-Routine die Ausführung unterbrochen, da der Kernel evtl. auf Ressourcen für den Thread wartet. Bevor *i386_do_context_switch* aufgerufen wird, speichert *i386_yield_kernel_thread* den Betriebszustand des Kernels an der Stelle auf dem Kernel-Stack, an der der Aufrufer von *SYSCALL_YIELDS_THREAD* seine Ausführung unterbrochen hat. Bei Wiederaufruf des Kernels wird statt eines Benutzermodus-Zustandes der Kernelzustand geladen.

Da der Zustand des Benutzermodus jedoch immer an fester Adresse liegt, können somit die Benutzermodus-Register eines Threads durch die Systemaufrufe *read_regs* und *write_regs* manipuliert werden - die Register eines im Kernel festsitzenden Threads jedoch nicht, da diese eben an anderer Stelle auf dem Kernel-Stack gespeichert sind. Bei Initialisierung eines neuen Kernel-Stacks (siehe 11.2.1) wird aus gleichem Grund am Ende des Kernel-Stacks ein Prototyp-Status gespeichert, der bei Aktivierung des Threads von *i386_do_context_switch* schließlich geladen wird.

Insgesamt ist dieser Mechanismus etwa genauso effektiv, wie der von Intel angebotene TSS-Mechanismus. Der *hymk* verwendet das TSS-Segment nur um das Stack-Register für den Ring-0 (*ESP0*) für den Eintritt in den Kernel bereitzulegen, wie es die CPU-Spezifikation fordert. Der eigentliche CPU-Zustand wird im TSS nicht gespeichert. Es existiert daher im übrigen nur ein einziges TSS-Segment, dass durch die Datenstruktur *i386_tss_struct* (*tss.c*) beschrieben wird[4].

12.4 Der Scheduler

12.4.1 Der Scheduling-Algorithmus

Der Scheduling-Algorithmus des *hymk* ist recht primitiv[5]. Es handelt sich um einen prioritätsbasierten Scheduler nach Round-Robin-Prinzip mit zusätzlicher Bevorzugung von I/O-intensiven Threads. Der Scheduler unterscheidet zwischen zwei Arten von Thread-Prioritäten: Die statische und die effektive Priorität.

Prioritäten

Die statische Priorität wird durch den Systemaufruf *set_priority* durch einen berechtigten Prozess festgelegt. Sie besteht aus Zahlen zwischen 0 und 40, wobei 40 der höchsten Priorität entspricht. Normale Threads dürfen ihre statische nur absenken. Threads von Root-Prozessen können sie auch anheben.

Die effektive Priorität hingegen wird jedesmal errechnet, wenn ein Thread aus einer I/O-Operation aufwacht oder wieder regulär nach der Warteschlange (die sog. *Runqueue*) zur Ausführung kommt. Die effektive Priorität gibt zugleich an, wieviele Uhrenticks (1 ms) ein Thread ausgeführt werden darf,

[3] Ausgenommen sind hierbei sämtliche FPU-Register, da diese an anderer Stelle wiederhergestellt werden.

[4] Nähere Informationen zum Aufbau des TSS-Segments können der IA-32-Dokumentation entnommen werden.

[5] Vermutlich wird jeder etwas besser gebildete Akademiker dieses gebilde nicht Scheduler nennen wollen.

ehe er die CPU-Zeit verliert. Kommt ein Thread normal an die Reihe entspricht dies auf x86-Systemen dem doppelten Zahlenwert der statischen Priorität. Wacht jedoch ein Thread aus einer I/O-Operation auf, so wird ihm zusätzlich noch die Hälfte der effektiven Priorität gutgeschrieben, die noch übrig war, als er die CPU auf Grund der I/O-Blockade abgegeben hatte.

Scheduling-Policies

Daneben gibt es theoretisch auch mehrere Scheduling-Klassen (*Policies*), mit deren Hilfe zwischen unterschiedlichen Scheduling-Verfahreng gewechselt werden kann. Derzeit ist aber nur eine verfügbar, *SCHED_REGULAR* (*sched.h*). Die Scheduling-Klassen sind bestimmten numerischen Werten zuzuweisen. Bei Änderung der Klasse gilt, dass Threads normaler Prozesse nur Klassen mit Nummern auswählen können, die kleiner oder gleich dem aktuellen Klassenwert sind. Root-Prozesse hingegen können auch größere Werte verwenden. Dadurch soll später auch eine Priorisierung der Scheduling-Policy vorgenommen werden können (z.B. Echtzeit vs. Normalbetrieb). Derzeit existiert jedenfalls nur eine Klasse mit Nummer 0.

Die Runqueue

Die wichtigste Datenstruktur des Schedulers ist die Runqueue. In ihr sind alle Threads enthalten, die derzeit ausgeführt werden sollen. Sie ist genaugenommen eine verkette Liste, die jeweils über die Elemente *THRTAB_RUNQUEUE_PREV* und *THRTAB_RUNQUEUE_NEXT* der jeweiligen Thread-Deskriptoren verknüpft ist. Den Beginn der Liste bildet immer der sog. *idle*-Thread - alle anderen Threads folgen ihm. Der Deskriptor des aktuellen Threads wird durch die Variable *current_t* (*current.c, current.h*) angezeigt.

Der *idle*-Thread

Dieser Thread läuft stets im Kernel-Mode und dient dazu, die CPU mit energiesparenden HLT-Instruktionen zu versorgen (siehe *ksched_idle_loop, schedule.c*), wenn alle Threads auf Eingaben warten. Außerdem frisst er rein formal die übrige CPU-Zeit auf, solange alle Threads blockiert sind. Auf den Deskriptor dieses *idle*-Thread ist im Kernel in der Variable *ksched_idle_thread* (*schedule.c, sched.h*) ein Zeiger abgelegt.

Umsetzung der Prioritätenregel

Normalerweise wird ein Thread einfach solange ausgeführt, bis seine CPU-Zeit abgelaufen ist. Anschließend wird der nächste Thread in der Liste ausgewählt.

Wird jedoch während dieser Ausführungszeit durch ein Ereignis (IRQ oder Aufhebung einer Blockade durch einen Systemaufruf) ein anderer Thread mit höherer effektiver Priorität aufgeweckt, so wird dieser Thread als Nachfolger des aktuellen Threads in die Runqueue eingefügt und ein Threadwechsel forciert. Dadurch soll eine kürzere Antwortzeit bei I/O-Operationen erreicht werden, da i.d.R. Threads, die auf Eingaben warten noch verbleibende Restprioritäten haben und diese bei Reaktivierung auf Grund der eintreffenden Eingabe gutgeschrieben wird. Dies ist sinnvoll, um dem Anwender einen befriedigenden Eindruck von Reaktionszeit und Geschwindigkeit des Systems zu liefern.

12.4.2 Verwaltung der Runqueue

Aufnahme in die Runqueue

Zur Aufnahme von Threads in die Runqueue ist die Kernel-interne Funktion *ksched_start_thread* (*schedule.c, sched.h*) zuständig. Diese Funktion prüft selbstständig, ob ein Thread bereits Mitglied der Queue ist und in wiefern er überhaupt aufgenommen werden darf. Darf er aufgenommen werden,

wird von der Funktion die effektive Priorität ausgerechnet. Ist die effektive Priorität des neuen Threads größer als die Priorität des aktuellen, so wird der Thread direkt nach dem aktuellen Thread in die Runqueue eingereiht und ein Threadwechsel forciert.. Andernfalls wird der Thread an den Anfang der Runqueue direkt nach dem *idle*-Thread eingereiht[6].

Anschließend wird das Flag *THRSTAT_BUSY* aus dem Thread-Status *THRTAB_THRSTAT_FLAGS* entfernt, so dass der Thread wieder als Betriebsbereit gilt. Ebenfalls wird der Zähler für aktive Threads *ksched_active_threads* (*schedule.c*, *sched.h*) inkrementiert.

Entfernen aus der Runqueue

Soll ein Thread aus der Runqueue entfernt werden, muss der Aufruf *ksched_stop_thread* (*schedule.c*, *sched.h*) getätigt werden. Dieser Aufruf entfernt den Thread aus der Runqueue und setzt dessen Felder *THRTAB_RUNQUEUE_PREV* und *THRTAB_RUNQUEUE_NEXT* auf Null. Der Zähler *ksched_active_threads* wird dekrementiert und das *THRSTAT_BUSY*-Flag gesetzt.

Dieser Aufruf forciert keinen Threadwechsel durch Änderung von *ksched_change_thread*, falls der zu stoppende Thread gleich dem aktuellen Thread ist. Die aufrufende Routine hat dies zu erledigen.

12.4.3 Forcieren eines Threadwechsels

Um das System zur Ausführung eines Threadwechsel zu veranlassen, muss einfach nur die Variable *ksched_change_thread* (*schedule.c*, *sched.h*) auf 1 gesetzt werden und der Aufruf *ksched_next_thread* (*current.c*, *current.h*)[7] ausgeführt werden. Soll davor implizit mit Hilfe von *ksched_change_thread* überprüft werden, ob ein Thread-Wechsel überhaupt nötig ist, kann dies über das Makeo *KSCHED_TRY_RESCHE* gemacht werden. Ist ein Thread-Wechsel nötig, führt es diesen aus.

Der Aufruf *ksched_next_thread* führt alle Vorbereitungen für einen Thread- und ggf. Prozesswechsel zum nächsten Thread der Runqueue durch, so dass beim Ausführen von *i386_do_context_switch* (*irq.s*, siehe 12.3) der Kernel dann beim Wechsel in den Benutzermodus einen Threadwechsel vollzieht. Voraussetzung ist - wie bereits erwähnt -, dass *ksched_change_thread* zuvor auf 1 gesetzt wurde.

Sollte *ksched_active_threads* den Wert 1 haben, wird hierbei stets automatisch der *idle*-Thread als nächster Thread ausgewählt. Andernfalls der nächste Thread der Liste - oder bei erreichtem Listenende - der erste Thread nach dem *idle*-Thread.

Die Argumente des Aufrufs *i386_do_context_switch* - die Zeiger *i386_new_stack_pointer* und *i386_old_stack_pointer* - werden von diesem Aufruf auf die jeweiligen Stackpointer-Puffer im Deskriptor des Quell- (*_old_*) bzw. Ziel-Threads (*_new_*) gesetzt.

Ist der zu startende Thread Teil eines anderen Prozesses, wird über den Aufruf *ksched_switch_space* (*current.h*) ein Kontextwechsel durchgeführt. Dabei wird u.a. der Zeiger *i386_current_pdir*, der das aktuelle Seitenverzeichnis anzeigt, auf das Seitenverzeichnis des neuen Prozesses (aus *PRCTAB_PAGEDIR_PHYS* geändert. Der Wechsel findet nach den IA-32-Spezifikationen dadurch statt, dass die Adresse des neuen Seitenverzeichnisses in das *CR3*-Register der CPU geladen wird. Da der Kernel-Adressraum dabei konstant bleibt, kann die Ausführung danach normal weiterlaufen. Ebenfalls wird danach in *ksched_next_thread* der Zeiger *current_p*, der auf den Deskriptor des aktuellen Prozesses verweist, entsprechend geändert und die Haupt-Info-Page aktualisiert. Der Aufruf *ksched_switch_space* prüft aus Performance-Gründen nicht die Zulässigkeit des genannten Prozesses.

Nachdem der Adressraum gewechselt wurde, werden alle FPU (etc.)-Register über die Funktion *ksched_switch_fpu_state* (*current.c*) geladen. Diese Information ist im Feld *THRTAB_X86_FPU_STACK*

[6] Etwaige Benachteiligungen anderer Thread dadurch werden derzeit dabei vernachlässigt. Eine Alternative wäre, den neuen Thread direkt vor den aktuellen Thread einzureihen, so dass er wirklich erst nach allen anderen Drankommt. Es könnte natürlich auch eine Einreihung nach Priorität durchgeführt werden.

[7] Anmerkung: Es könnte evtl. später einmal effizienter sein, diesen Aufruf als *inline*-Funktion zu realisieren.

des jeweiligen Threads gespeichert. Der Kernel speichert den Registerzustand des verdrängten Threads über die *fsave*-Instruktion oder - falls verfügbar - über die *fxsave*-Instruktion dort ab und lädt aus dem gleichen Feld des Zielthreads dessen FPU-Register mittels *frstor* bzw. *fxrstor* neu ein, so dass diese nach Reaktivierung des Threads wieder zur Verfügung stehen [8].

Als nächstes wird der Zeiger *current_t*, der auf den Deskriptor des aktuellen Threads verweist geändert. Anschließend wird über über den Aufruf von *ksched_set_local_storage* der *thread local storage* des neuen Threads eingeblendet, wobei dieser Aufruf sich *kmem_map_page_frame_cur* bedient.

Nach Aktualisierung der Haupt-Info-Page wird der Zegier auf den Deskriptoreintrag über die effektive Priorität des neuen Threads nach *kinfo_eff_prior* geladen. Diese Variable wird später verwendet, um die effektive Priorität währernd des Systembetriebs bei jedem IRQ der Systemuhr herabzusetzen. Hat der Thread bei verlassen von *ksched_next_thread* keine effektive Priorität, so wird die Priorität aus der statischen Priorität des Threads errechnet.

Ebenfalls wird zur Anpassung der I/O-Zugriffsrechte beim Threadwechsel der Zeiger *kinfo_io_map* auf das Informationsfeld zu den I/O-Zugriffsrechten im Deskriptor des nun aktuellen Prozesses gesetzt (siehe 12.2.2).

Zu beachten ist, dass *ksched_next_thread* immer nur am Schluß einer Operation ausgeführt werden sollte, wenn der Threadwechsel wirklich ausgeführt werden soll, da verschiedene interne Zeiger (u.a. der auf den aktuellen Thread) dann bereits auf den neuen Thread eingestellt sind.

Grundsätzlich wird ein durch *ksched_next_thread* forcierter Threadwechsel bei jedem Austritt aus dem Kernel wirksam. Egal, ob das System durch einen IRQ, eine Exception oder einen Systemaufruf in den Kernel eingetreten ist. Ebenfalls ist egal, ob ein Systemaufruf normal terminiert hat oder seine Ausführung durch *SYSCALL_YIELDS_THREAD* nur einstweilen unterbrechen will.

12.4.4 Eintritt in die Schedulingschleife

Nach vollständiger Initialisierung des Systems, tritt der Kernel in die Scheduling-Schleife ein. Hierfür ist die Funktion *ksched_enter_main_loop* (*current.c*, *sched.h*) zuständig. Diese Funktion führt *ksched_next_thread* mit gesetztem *ksched_change_thread* aus, so dass ein Threadwechsel forciert wird. Da der Kernel nach dem Start einen temporären Stack verwendet, wird *i386_old_stack_pointer* nachträglich auf einen internen Zeiger geändert, da der Kernel-Stack des *idle*-Threads nicht verändert werden darf.

Der Eintritt in die Scheduling-Schleife erfolgt letztlich durch einen Sprung nach *i386_do_context_switch*. Da der neue Zielthread mit aktivierten IRQs arbeitet, wird beim ersten Eintritt in den Benutzer-Modus automatisch die IRQ-Behandlung aktiviert.

12.4.5 Die Initialen Prozesse

Der *hymk* startet automatisch zwei sog. initiale Prozesse, die für den Betrieb des Systems nach der Initiailsierung des Kernels benötigt werden. Der eine ist der bereits erwähnte idle-Prozess - der andere ist der sog. Init-Prozess, der die Initialisierung des Systems übernimmt.

Der idle-Prozess

Wie bereits in Abschnitt 12.4.1 erläutert, dient der *idle*-Thread dazu, die CPU in einen Wartezustand zu versetzen, wenn kein Thread aktiv ist. Diesem *idle*-Thread ist ein Prozess zugeordnet. Auf x86-Plattformen trägt der Thread die SID 0x1000000 und der Prozess die SID 0x2000000.

[8] Hier sollte geprüft werden, inwiefern die Speicherung der Registerdaten im öffentlich lesbaren Thread-Deskriptor ein Sicherheitsrisiko darstellen könnte. Als Alternative könnten die Daten am unteren Ende des Kernel-Stacks ebenfalls gesichert werden.

Initialisiert wird der *idle*-Prozess durch den Kernel in der Routine *ksysc_create_idle* (*subject.c*). Da diese Funktion den ersten Prozess des Systems erstellt, muss sie *current_p* und *current_t* im Voraus initialisieren, wobei diese Zeiger auf die Adresse der zukünftigen Deskriptoren von *idle* gesetzt werden. Der neue Prozess selbst (und somit auch dessen initialer Thread) wird über den Aufruf von *sysc_create_process* ganz konventionell erstellt. Dabei wird auch die Variable *ksched_idle_thread* (*schedule.c, sched.h*) initialisiert.

Da der Idle-Thread im Kernel-Modus operieren soll, muss dessen Kernel-Stack mit gesetztem Flag *KSCHED_KERNEL_MODE* erneut initialisiert werden. Als Startroutine erhält der Idle-Thread die bereits erwähnte Routine *ksched_idle_loop* (*schedule.c, sched.h*).

Die Priorität wird des *idle*-Threads wird anschließend auf das niedrigste Niveau herabgesetzt. Der Thread wird daraufhin durch *sysc_awake_subject* aktiviert, so dass er beim Eintrtt in die Schedulingschleife zur Ausführung verfügbar wird.

Wichtig ist, dass der *idle*-Thread vom Benutzermodus aus nicht kontrolliert oder beeinflusst werden können darf. Der *idle*-Thread operiert ständig im Kernel-Modus und ist für den Multitaskingbetrieb unerlässlich. Eine Eingriffsmöglichkeit aus dem Benutzermodus stellt ein hohes Stabilitäts- und Sicherheitsrisiko dar.

Der init-Prozess

Nachdem der Kernel soweit initialisiert wurde, muss ein Programm gestartet werden, das den Rest des Betriebssystems lädt. Dieses Programm heißt unter HydrixOS *init* und wird vom Bootloader beim Start als Image in einen Speicherbereich geladen, der dem Kernel zugänglich ist. Damit der Kernel das richtige Modul ermitteln kann, wird von einem Bootloader, der der Multiboot-Spezifikation entspricht (im wesentlichen ist das GRUB), eine Tabelle mit Informationen über die vorhandenen Module hinterlassen. Im *hymk* ist diese Tabelle über den Zeiger *grub_modules_list* (*modules.c, mem.h*) erreichbar, wobei *init* dort im Eintrag 0 enthalten ist.

Der Kernel lädt und initialisiert *init* über die Funktion *ksysc_create_init* (*subject.c*). Diese Funktion erzeugt einen Prozess über den Aufruf *sysc_create_process*. In den Adressraum des Prozesses werden die Daten des Init-Prozesses über *kmem_map_page_frame* (*page.c, mem.h*) eingeblendet. Anschließend wird dem Prozess die maximale Priorität verliehen und durch *sysc_awake_subject* (*schedule.c, sysc.h*) gestartet. Da der Init-Prozess ein Tochterprozess des Idle-Prozesses ist, erbt er volle Root- und I/O-Rechte.

Der Aufruf von *ksysc_create_init* kann prinzipiell Mehrfach bei der Initialisierung des Kernels aufgerufen werden, um z.B. den Kernel tiefergehend zu testen. Nach dem Aufruf von *ksysc_create_init* ist als aktueller Adressraum der virtuelle Adressraum des erzeugten Prozess gesetzt. Da durch *awake_subject* ein hochprioritärer Thread erzeugt wurde, ist zu erwarten, dass dieser bei Eintritt in die Scheduling-Schleife gestartet wird.

12.5 Behandlung von Timeouts

Synchrone Operationen erfordern es meistens, dass eine Thread auf einen anderen Thread warten muss. Bei diesen synchronen Operationen (im *hymk* sind dies im wesentlichen *recv_softint* und *sync*), können Timeouts gesetzt werden, nach deren Ablauf der Kernel die Operation abbrechen soll.

12.5.1 Verwendetes Verfahren

Der Kernel bietet hierfür ein internes Verfahren an, das von den verschiedenen Systemaufrufen aufgegriffen werden kann. Dieses Verfahren baut auf einer Warteschlange auf, in der in Reihenfolge der verschiedenen Timeouts alle wartenden Threads aufgelistet sind. Der jeweils erste Thread dieser Warteschlange ist der Thread, der als nächstes aufgeweckt werden soll. Um unnötiges Herunterzählen

von Zeit zu vermeiden, wird nicht die verbleibende Zeit des Timeouts, sondern der Zeitpunkt relativ zur verstrichenen Zeit seit dem Systemstart notiert. Wird der Zeitpunkt erreicht, wird das Timeout erreicht.

Die Liste benötigt keinen besonderen Speicherplatz, sondern ist über die Thread-Deskriptoreneinträge *THRTAB_TIMEOUT_QUEUE_NEXT* und *THRTAB_TIMEOUT_QUEUE_PREV* untereinander verknüpft. Der erste Thread der Liste ist über die Variable *timeout_queue* (*timeout.c*, *sched.h*) erreichbar. Die Anzahl der Threads in der Warteliste kann über *timeout_num* (*timeout.c*, *sched.h*) ermittelt werden. Der Zeitpunkt, an dem der nächste Thread sein Timeout erreicht, ist in der Variable *timeout_next* (*timeout.c*, *sched.h*) gespeichert.

Die Liste kann über die Funktionen *ksched_add_timeout* (*timeout.c*, *sched.h*) und *ksched_del_timeout* (*timeout.c*, *sched.h*) bearbeitet und verwaltet werden. Beide Funktionen verwalten lediglich die Timeout-Liste und ändern den Timeout-Wert eines Threads. Sie halten aber weder den Thread an, noch setzen sie das Flag *THRSTAT_TIMEOUT* - dies ist Aufgabe der aufrufenden Funktionen.

12.5.2 Format eines Zeitpunkts

Jeder Zeitpunkt wird als Relativzeit zum Systemstart in Millisekunden gespeichert. Der Zeitpunkt des Timeouts eines Threads wird in den Deskriptoreinträgen *THRTAB_TIMEOUT_LOW* und *THRTAB_TIMEOUT_H* gespeichert. Hier ist - wie auch bei der Systemzeit - eine Aufteilung in zwei 32-bit Werte erforderlich, da die Zeit als 64-bit Wert gespeichert wird, um einen Uhren-Überlauf unwahrscheinlich zu machen (das System kann bei einem 64-bit Wert über 580 Mio. Jahre sich im Dauerbetrieb befinden).

Ein Timeout selbst kann aber nur maximal 32-bit (2^32 ms - d.h. 49 Tage) lang sein. Ein Timeout darf nicht den Wert *0* (kein Timeout) und nicht den Wert *0xFFFFFFFF* (*unendlich*) haben. Solche Timeouts werden zwar von *ksched_add_timeout* akzeptiert, werden aber rein numerisch behandelt - ihre Sonderbedeutung wird dabei ignoriert. Unendliche Timeouts müssen daher von der aufrufenden Routine bereits richtig behandelt werden.

12.5.3 Aufnahme in die Warteschlange

Die Funktion *ksched_add_timeout* (*timeout.c, sched.h*) erhält vom Aufrufer das Timeout in Millisekunden und errechnet selbstständig über die Systemzeit (gespeichert in *kinfo_rtc_ctr*, *info.c*, *info.h*) den Zeitpunkt, andem das Timeout endet. In *THRTAB_TIMEOUT_LOW* und *THRTAB_TIMEOUT_HIGH* wird dann der Zeitpunkt gespeichert.

Die Routine unterscheidet zwischen vier unterschiedlichen Arten, ein Timeout zur Warteschlange hinzuzufügen: Entweder war die Warteschlange leer (*timeout_queue == NULL*) und somit wird die Warteschlange neu initialisiert und als nächstes Timeout (*timeout_next*) das, des zu installierenden Threads eingerichtet.

Oder aber das Timeout des zu installierenden Threads endet vor dem des ersten Threads der Liste - hierbei wird der neue Thread auf den Anfang der Liste gelegt und der Zeitpunkt des nächsten Timeouts entsprechend geändert.

Die dritte Möglichkeit ist, dass der Thread vor einem anderen Thread der Liste drankommen kann. Hierfür wird die komplette Liste durchsucht. Wird ein Thread gefunden, dessen Timeout erst nach dem neuen Thread erreicht ist, wird der neue Thread vor diesem in die Liste eingefügt. Zuletzt kann ein Thread auch noch an das Ende der Liste gestellt werden und die Liste terminieren.

12.5.4 Entfernung aus der Warteschlange

Wird ein Timeout erreicht oder aus einem anderen Grund das Warten abgebrochen (z.B. bei Erreichen des gewünschten Ereignisses), kann der Thread mittels *ksched_del_timeout* wieder aus der Liste entfernt werden. Zur Beschleunigung des Verfahrens unterscheidet die Routine zwischen zwei Fällen:

Ist der zu entfernende Thread der Thread am Anfang der Liste, so wird dessen Nachfolger einfach nachgerückt - oder bei erreichtem Listenende die Liste geschlossen. Ist der zu entfernende Thread mitten in der Liste, wird er aus der Liste heraus entfernt.

Beim Beenden eines Threads wird dieser Automatisch von *sysc_destroy_subject* aus der Timeout-Warteschlange entfernt.

12.5.5 Prüfung der Timeouts

Ob der Thread am Anfang der Timeout-Warteschlange aufgeweckt werden soll, oder nicht, wird bei jedem Uhren-IRQ durch die Funktion *ksched_handle_irq* überprüft. Da das Timeout in der Variable *timeout_next* (*timeout.c*, *sched.h*) gespeichert wird, wird ein unnötiger Zugriff auf den Thread-Deskriptor vermieden.

Wurde das Timeout erreicht, wird der Thread mittels *ksched_del_timeout* (*timeout.c*, *sched.h*) aus der Warteschlange entfernt und durch *ksched_start_thread* (*schedule.c*, *sched.h*) wieder neu gestartet. Ebenfalls wird das Flag *THRSTAT_TIMEOUT* wieder gelöscht.

12.6 IRQ-Behandlung

12.6.1 Ablauf einer IRQ-Behandlung

Ein IRQ wird durch ein externes Hardwareereignis ausgelöst. Das auslösende Gerät sendet an den Interrupt-Controller (PIC) der CPU ein Signal. Der PIC wiederum sperrt den IRQ und übermittelt der CPU ein entsprechendes Signal, das sie zum Unterbrechen der aktuellen Ausführung und Behandlung des Interrupts auffordert. Die Behandlung des IRQs auf niedrigster Ebene wurde bereits im Kapitel über den Kernel-Eintritt (siehe 12.1.3) angesprochen. Wie aber nun vom Kernel aus die IRQ-Information an den Thread eines Gerätetreibers weitergegeben wird, wird nun in diesem Kapitel geklärt.

12.6.2 Der Systemaufruf *recv_irq*

Ein Thread, der auf einen IRQ warten will, muss den Systemaufruf *recv_irq* - impelementiert durch *sysc_recv_irq* (*io.c*, *sysc.h*) ausführen. Dieser Systemaufruf lässt den Thread passiv auf den IRQ warten. Tritt er ein, wird die Ausführung des Threads mit hoher Priorität fortgesetzt. Nach der Behandlung kann der Thread erneut auf den gleichen IRQ warten oder aber den IRQ anderen Threads wieder zur Verfügung stellen.

Es ergeben sich also für den Systemaufruf zwei Aufgabenbereiche, die nun im folgenden geklärt werden:

1. Warten auf einen IRQ

Wartet ein Thread das erste mal auf einen bestimmten IRQ, muss dieser zuerst durch *recv_irq* reserviert werden. Zur Reservierung von IRQs existiert die Datenstruktur *irq_handlers_s* (*io.c*), die folgenden Aufbau hat:

```
struct
{
    sid_t tid;
    int is_handling;
}irq_handlers_s[16];
```

Jeder Eintrag des Arrays entspricht einem der sechzehn IRQs der x86-Plattform. Das Unterelement *tid* gibt die Thread-SID des Threads an, der den IRQ derzeit belegt (ist der Wert 0, so gilt der IRQ als nicht belegt. Das Unterelement *is_handling* hingegen gibt an, ob derzeit darauf gewartet wird, ob der IRQ eintritt (0) oder ob der IRQ derzeit vom Thread *tid* behandelt wird (1).

Soll ein IRQ nun reserviert werden, wird zuerst geprüft, ob die IRQ-Nummer gültig ist und ob der IRQ bereits belegt ist. Ist dem nicht so, wird der entsprechende Eintrag in *irq_handlers_s* konfiguriert. Dementsprechend wird ebenfalls der Deskriptor des aufrufenden Threads geändert. Dem Element *THRTAB_IRQ_RECV_NUMBER* wird die Nummer des aktuellen IRQs zugewiesen. Als effektive Priorität erhält der Thread *IRQ_THREAD_PRIORITY*, was normalerweise ausreichend hoch seien sollte.

Anschließend wird der Thread durch *ksched_stop_thread* (*schedule.c*, *sched.h*) gestoppt und der IRQ über *ksched_enable_irq* (*intr.c*, *sched.h*) freigegeben. Diese Funktion führt plattformspezifische Operationen aus, die den Interrupt-Controller anweisen, den IRQ an die CPU weiterzuleiten.

Zum Schluß wird ein Threadwechsel durch setzen von *ksched_change_thread* (*schedule.c*, *sched.h*) und Ausführen von *ksched_next_thread* (*current.c*, *current.h*) forciert.

Beim erneuten Warten auf einen IRQ passiert im Prinzip das gleiche, wie beim erstmaligen Warten auf einen IRQ. Einziger Unterschied ist, dass der IRQ bereits durch den aktuellen Thread reserviert ist. Wird jedoch auf einen anderen IRQ gewartet, als im Wartevorgang zuvor, wird zuerst der IRQ wieder freigegeben

2. Freigabe eines IRQs

Ein IRQ wird freigegeben, wenn auf den IRQ *0xFFFFFFFF* oder wenn auf einen neuen, anderen IRQ gewartet werden soll. Hierfür ist die Routine *kio_reenable_irq* zuständig, die von *sysc_recv_irq* in einem der beiden Fällen aufgerufen wird.

Diese Routine setzt den *irq_handlers_s*-Eintrag wieder auf 0 zurück und setzt den Eintrag *THRTAB_IRQ_REC* des Thread-Deskriptors wieder auf *0xFFFFFFFF*.

Der IRQ selbst wird mittels *ksched_disable_irq* (die wieder einige x86-Spezifische Operationen ausführt, um den IRQ zu sperren; siehe *intr.c*, *sched.h*) gesperrt. Der IRQ wird nicht gesperrt, wenn es sich um den IRQ der Systemuhr (IRQ 0) handelt, da dieser IRQ vom Kernel für den Scheduler verwendet wird.

Wenn der Thread keinen weiteren IRQ mehr behandeln will, verliert er seine effektive Priorität und wird zu einem Threadwechsel gezwungen, um anderen Threads nicht ungerechtfertigt zu benachteiligen.

12.6.3 Der Weg vom Kernel zum Thread

Tritt ein IRQ ein, ruft die Low-Level-Routine des Kernels die Funktion *ksched_handle_irq* (*intr.c*) auf. Diese Routine erhält als Parameter die Nummer des aufgerufenen IRQs. Stellt die Routine fest, dass es sich um den IRQ 0 handelt, so erhöht sie den Zähler der Uhr (*kinfo_rtc_ctr*) [9]. Außerdem prüft sie, ob der aktuelle Thread sein Zeitquantum aufgebraucht hat oder nicht. Hat er es aufgebraucht, wird ein Threadwechsel forciert. Andernfalls wird seine effektive Priorität (und somit sein Zeitquantum) verringert. Der Zeiger *kinfo_eff_prior*, der beim Threadwechsel zuvor gesetzt wurde, erlaubt hierbei einen schnellen Zugriff auf den entsprechenden Eintrag im Threaddeskriptor.

In jedem Fall wird der IRQ an die Routine *ksched_start_irq_handler* (*io.c*, *sched.h*) weitergegeben. Diese Routine prüft, ob ein Thread auf diesen IRQ wartet bzw. ob er derzeit schon behandelt wird. Wartet ein Thread auf diesen IRQ und wird er noch nicht behandelt, so wird der *is_handling*-Eintrag in der *irq_handler_s*-Datenstruktur gesetzt und der IRQ mit *ksched_disable_irq*

[9] Bei einem Überlauf der Uhr wird eine Kernel-Panic ausgelöst, da anhand der langen Laufzeit der Uhr (mehrere Mio. Jahre) eher anzunehmen ist, dass die Systemuhr defekt ist.

(*intr.c, sched.h*) deaktiviert. Auch hier ist der IRQ 0 davon ausgenommen. Eine Mehrfachbehandlung durch einen Thread wird hierbei durch das *is_handling*-Flag vermieden.

Der behandelnde Thread wird nun über *ksched_start_thread* (*schedule.c, sched.h*) ganz normal wieder in die Runqueue aufgenommen. Es wird ein Threadwechsel forciert und da der Thread eine sehr hohe effektive Priorität hat, wird er höchstwahrscheinlich bei Rückkehr aus dem Kernel-Modus gestartet werden.

12.6.4 Schlußbemerkungen zur IRQ-Behandlung

Der Kernel befindet sich also während der Behandlung eines IRQs durch einen Treiber in keinem besonderen Betriebsmodus. Der IRQ wird direkt weitergegeben und der Treiber ganz normal im Multitasking-Zyklus ausgeführt. Ist während der IRQ-Behandlung das Multitasking unerwünscht, muss der jeweilige Treiber den Interruptcontroller anweisen, den IRQ 0 zu sperren.

Der Kernel muss dies berechtigten Threads gestatten und ebenfalls genug Zeit dem behandelnden Thread beim Beginn der IRQ-Behandlung durch eine entsprechend hohe effektive Priorität einräumen.

12.7 Systemaufrufe zur Ausführungskontrolle

12.7.1 Der Systemaufruf *set_priority*

Für die Änderung der statischen Priorität ist der Systemaufruf *set_priority*, implementiert durch den Aufruf *sysc_set_priority* (*schedule.c, sysc.h*) zuständig. Der Systemaufruf prüft lediglich die Zugriffsrecht und ändert die Daten der statischen Priorität und Scheduling-Klasse. Bei erneuter Berechnung der effektiven Priorität wird die neue statische Priorität beachtet.

12.7.2 Der Systemaufruf *freeze_subject*

Soll ein Thread eingefroren werden, muss der Systemaufruf *freeze_subject* getätigt werden. Dieser Systemaufruf - implementiert durch *sysc_freeze_subject* (*schedule.c, sysc.h*) - ist derzeit nur auf Thread-Subjekte beschränkt, kann aber in späteren Versionen auch auf andere Subjektarten ausgeweitet werden.

Der Aufruf selbst erhöht (nach einer ausführlichen Prüfung der Zugriffsrechte), den Freeze-Counter des Threads und setzt das Flag THRSTAT_FREEZED. Anschließend wird geprüft, ob der Thread bereits eingefroren ist, sei es durch einen vorherigen Freeze-Aufruf (Freeze-Counter ist erhöht) oder aus einem anderen Grund (eines der unter THRSTAT_OTHER_FREEZED zusammengefassten Bits ist gesetzt). Ist der Thread noch nicht eingefroren gewesen, wird er aus der Runqueue mittels *ksched_stop_thread* entfernt. Handelt es sich zudem um den aktuellen Thread, wird ein Threadwechsel forciert.

12.7.3 Der Systemaufruf *awake_subject*

Ein bereits eingefrorener Thread kann über den Systemaufruf *awake_subject* wieder aufgeweckt werden. Der Systemaufruf ist über *sysc_awake_subject* (*schedule.c, sysc.h*) implementiert. Auch hier ist eine Erweiterung auf andere Subjekt-Arten in späteren Versionen möglich.

Dieser Aufruf dekrementiert den Freeze-Counter des Threads. Ist dieser bei 0 angelangt, wird der Thread über *ksched_start_thread* (*schedule.c, sched.h*) wieder gesartet, sofern nicht der Thread zuvor schon aus einem anderen Grund eingefroren wurde (eines der unter THRSTAT_OTHER_FREEZED zusamengefassten Bits ist gesetzt). Da der neu gestartete Thread möglicherweise eine höhere effektive Priorität als der aktuelle Thread haben kann, wird mit *ksched_next_thread* ggf. ein forcierter Threadwechsel gültig gemacht.

12.7.4 Der Systemaufruf *yield_thread*

Ein Systemaufruf hat die Möglichkeit seine CPU-Zeit freiwillig abzugeben. Hierfür ist der Systemaufruf *yield_thread*, implementiert durch *sysc_yield_thread* zuständig. Im wesentlichen setzt dieser Aufruf die effektive Priorität des Aufrufers auf 0 und forciert durch Setzen von *ksched_change_thread* und *ksched_next_thread* einen Threadwechsel. Der Systemaufruf erlaubt auch, die effektive Priorität an einen anderen Thread weiterzugeben, was lediglich durch Addition der Priorität zu der des anderen Threads geschiet.

12.8 Der *sync*-Mechanismus

Der einzige universelle Mechanismus für synchrone Kommunikation, der vom *hymk* angeboten wird, ist der *sync*-Mechanismus. Dieser Mechanismus wird durch den Systemaufruf *sync*, implementiert in *sysc_sync* (*sync.c*, *sysc.h*), zur Verfügung gestellt.

Hierbei ist eigentlich zwischen zwei Arten von Synchronisationen zu unterscheiden: Der direkten Synchronisation mit einem bekannten Thread-Subjekt und der indirekten Synchronisation mit einem beliebigen Subjekt. Beide Synchronisationen können sowohl aktiv einen anderen wartenden Thread aufwekcken oder zu passiven Wartezuständen des Aufrufers führen.

12.8.1 Grundlegende Datenstrukturen

Die wohl wichtigsten Datenstrukturen des Sync-Mechanismus sind die sog. Synchronisationswarteschleifen. Jeder Thread besitzt eine solche Warteschleife. Die Warteschleifen sind als verkettete Listen organisiert, wobei im Thread-Deskriptor ein Zeiger auf den ersten Thread von dessen Liste besteht (*THRTAB_OWN_SYNC_QUEUE_BEGIN*). Im Deskriptor eines Threads, der Mitglied einer Liste ist, verweisen die Elemente *THRTAB_CUR_SYNC_QUEUE_PREV* und *THRTAB_CUR_SYNC_QUEUE_NEXT* jeweils auf das vorherige bzw. nächste Element der Synchronisationsliste, in der er Mitglied ist.

12.8.2 Elemente des *sync*-Mechanismus

Da einige Elemente mehrfach genutzt werden und der Quelltext auch übersichtlich gehalten werden sollte, wurde der *sync*-Mechanismus auf weitere interne Funktionen verteilt, die hier nun erläutert werden (alle Funktionen sind im Modul *sync.c* enthalten):

ksync_is_waitqueue

Diese Funktion durchsucht die Synchronisationswarteschlange des aufrufenden Threads bei passiven Synchronisationen nach einem Thread, der den Suchkriterien der *sync*-Operation (siehe Beschreibung Systemaufruf *sync*, Kapitel 7.5.1) entspricht. Entweder wird von dieser Funktion die SID eines Threads zurückgegeben, der auf eine Synchronisation mit dem aktuellen Thread wartet. Oder aber es wird 0 zurückgegeben, wodurch die *sync*-Operation zum Start des passiven Wartevorgangs veranlasst wird.

ksync_wait_for

Diese Funktion startet einen passiven Wartevorgang. Ist ein Timeout gesetzt, wird das Timeout mittels *ksched_add_timeout* (*timeout.c*, *sched.h*) eingerichtet. Ist das Timeout 0 gesetzt, wird die Routine mit einer Fehlermeldung abgebrochen. In allen anderen Fällen (begrenztes oder unendliches Timeout) wird anschließend der Thread durch *ksched_stop_thread* (*schedule.c*, *sched.h*) gestoppt und durch *i386_yield_kernel_thread* verdrängt. Nach Reaktivierung des Threads durch einen anderen Thread,

startet die Funktion nach dem Yield-Befehl. Mittels des Aufrufs *MSYNC* wird sichergestellt, dass der Compiler nicht von einer ununterbrochenen Ausführung ausgeht.

Diese Funktion setzt ebenfalls das Flag *THRSTAT_SYNC*. Ist nach Ende des Wartevorgangs das Flag immer noch gesetzt, so fand keine Synchronisation statt, da der synchronisierende Thread dieses Flag löscht. Die Funktion terminiert dann mit einer Timeout-Fehlermeldung. Andernfalls gibt die Funktion die SID des Threads zurück, der sich mit dem Thread während der Wartezeit synchronisiert hat.

ksync_awake_other

Diese Funktion weckt einen anderen Thread auf, wenn dieser auf eine Synchronisierung mit dem aktuellen Thread gewartet hat. Ggf. entfernt die Routine den wartenden Thread aus der Timeout-Liste über *ksched_del_timeout* (*timeout.c, sched.h*) und startet ihn mittelst *ksched_start_thread* (*schedule.c, sched.h*). Wichtig ist, dass diese Funktion das Flag *THRSTAT_SYNC* beim wartenden Thread anschließend löscht und dessen Feld *THRTAB_SYNC_SID* auf die SID des aktuellen Threads anpasst, so dass dieser über den Erfolg der Synchronisierung informiert ist.

ksync_addto_waitqueue

Diese Funktion fügt den aktuellen Thread zur Synchronisationswarteschlange eines anderen Threads hinzu.

ksync_removefrom_waitqueue

Diese Funktion entfernt den aktuellen Thread aus der Synchronisationswarteschlange eines anderen Threads.

ksync_interrupt_other

Diese Funktion wird nicht von *sync*, sondern *destroy_subject* verwendet, wenn ein Thread vernichtet werden soll. Sie weckt einen auf den zu vernichtenden Thread wartenden Thread wieder auf. Da sie das Flag *THRSTAT_SYNC* gesetzt lässt, weiß der Thread nach dem aufwachen, dass die Synchronisierung fehlgeschlagen ist, so dass *sync* eine entsprechende Fehlermeldung an den Benutzermodus zurückreichen kann.

ksync_removefrom_waitqueue_error

Auch diese Funktion wird extern von *destroy_subject* verwendet, um einen zu vernichtenden Thread aus der Warteschlange eines anderen Threads zu entfernen.

12.8.3 Ablauf einer Sync-Operation

Direkte Synchronisation mit einem anderen Thread

Wenn sich ein Thread A direkt mit einem anderen Thread B synchronisieren will, so prüft *sync* zunächst, ob der B bereits auf den aktuellen Thread (A) wartet oder nicht. Wartet B bereits auf A, so wird B mit *ksync_awake_other* wieder aufgeweckt. Andernfalls trägt sich der Thread A in die Warteliste von Thread B durch *ksync_addto_waitqueue* ein und legt sich mittels *ksync_wait_for* schlafen, bis entweder das gesetzte Timeout endet oder ab B eine Synchronisation mit A startet und dabei A aufweckt. Nachdem A wieder aufgeweckt wurde, trägt sich A aus der Warteschlange von B durch *ksync_removefrom_waitqueue* aus und beendet den *sync*-Aufruf.

Indirekte Synchronisation mit einem Nicht-Thread-Subjekt

Wenn sich ein Thread mit einem Subjekt synchronisieren will, das kein Thread ist, so bleibt ihm nichts anderes übrig, als auf die Synchronisation mit Threads zu warten, die unter diese Subjekt-Kriterien (z.B. Prozess-SID) fallen. Bevor ein passiver Wartemodus begonnen wird, prüft jedoch der Thread durch den Aufruf von *ksync_is_waitqueue*, ob sich nicht möglicherweise in seine Synchronisationswarteschlange bereits ein Thread eingetragen hat, auf den diese Kriterien zutreffen. Ist dem so, so wird dieser mittels *ksync_awake_other* wieder aufgeweckt. Andernfalls wird ein passiver Wartezustand durch *ksync_wait_for* eingegangen, der entweder durch ein Timeout oder einen Thread beendet werden kann, der sich gezielt mit dem schlafenden Thread synchronisiert. Nach Ende des Wartevorgangs wird die Routine einfach beendet.

Implementierung des Resync-Mechanismuses

Die Resync-Fähigkeit des *sync*-Systemaufrufs wird durch eine einfache while-Schleife umgesetzt. Diese wird verlassen, wenn alle Resyncs getätigt worden sind oder bei einer Synchronisation ein Fehler oder ein leerer Rückgabewert aufgetreten ist.

Kapitel 13

Fernsteuerung von Threads

13.1 Lesen und Manipulieren von Registerinhalten

In Abschnitt 11.2.1 wurde bereits erläutert, dass der Anfangsbereich eines Kernel-Stacks stets die Registerdaten des Threads im Benutzermodus enthält. Aus diesem Grund kann das Auslesen und Manipulieren von Registerinhalten für die Systemaufrufe *read_regs* und *write_regs* leicht realisiert werden. Beide Systemaufrufe - jeweils implementiert über *sysc_read_regs* und *sysc_write_regs* (*remote.c*, *sysc.h*) - müssen lediglich direkt auf den Anfang des Kernel-Stacks eines Threads zugreifen, um die entsprechenden Registerdaten auszulesen oder zu ändern.

Eine Ausnahme bildet hier der *idle*-Thread, dessen Kernel-Stack den Registerzustand im Kernel enthält und daher von den Systemaufrufen nicht bearbeitet werden darf. Die FPU-Register der x86-Architektur werden derzeit zudem im Thread-Deskriptor und nicht im Kernel-Stack des Threads gespeichert, was mit der eigentümlichen Art und Weise zusammenhängt, mit der die x87-FPU ihre Registerinhalte speichert. Sie können mit *read_regs* weder ausgelesen, noch mit *write_regs* manipuliert werden[1].

13.2 Umleiten von Softwareinterrupts

Im Unterschied zu den meisten anderen Mikrokernen gestattet der *hymk* die Umleitung von Softwareinterrupts im Benutzermodus. Dies soll u.a. für die transparente Implementierung von Virtualisierungsdiensten ermöglichen. Hierfür zuständig ist der Systemaufruf *recv_softints*, der selbst wiederum durch den Aufruf *sysc_recv_softints* (*remote.c*, *sysc.h*) implementiert ist. Für dessen Umsetzung ist jedoch eine Zahl weiterer interner Aufrufe und Erweiterungen erforderlich.

13.2.1 Einleitung einer Überwachung

Beim Aufruf von *sysc_recv_softints* wird (nach der obligatorischen Sicherheitsprüfung und einem ggf. Aufwecken des Zielthreads) im Deskriptor des Zielthreads der Eintrag *THRTAB_SOFTINT_LISTENER_THR* auf die SID des Aufrufers gestellt. Dadurch werden später die verschiedenen Kernel-Einsprungroutinen prüfen können, ob eine Umleitugn eines aufgetretenen Softwareinterrupts erforderlich ist oder ob der Softwareinterrupt gewöhnlich abgearbeitet werden soll. Ebenfalls wird das Feld *THRTAB_RECEIVED_SOFTINT* im zu überwachenden Thread auf *0xFFFFFFFF* gesetzt, so dass hier später die Nummer des tatsächlichen Software-Interrupts eingetragen werden kann oder aber - bei bleibendem Wert *0xFFFFFFFF* ein Fehlschlag der Überwachung (z.B. bei Erreichen eines Timeouts) erkannt werden kann.

[1] Ein direktes auslesen über den Thread-Deskriptor ist hier allerdings auch für unberechtigte (nicht-root) Threads derzeit möglich. Da dies ein Sicherheitsproblem darstellen könnte, wird empfohlen, die FPU-Daten in späteren Versionen des hymk am unteren Ende des Kernel-Stacks zu speichern.

Im Deskriptor des Aufrufers wird das Flag *THRSTAT_RECV_SOFTINT* gesetzt und in das Element *THRTAB_RECV_LISTEN_TO* die SID des zu überwachenden Threads eingetragen. Anschließend wird ein passiver Wartemodus erreicht. Bei gesetztem Timeout wird hierbei die Timeout-Überwachung mittels *ksched_add_timeout* (*timeout.c, sched.h*) und durch Setzen des *THRSTAT_TIMEOUT*-Flags eingeschaltet. In jedem Fall wird der Thread anschließend durch *ksched_stop_thread* (*schedule.c, sched.h*) gestoppt und über den üblichen Weg verdrängt.

Durch das Makro *MSYNC* wird dem Compiler klar gemacht, dass ab dieser Position etwaige gechachte Variablen neu geladen werden müssen, da hier ein längerer passiver Wartezustand stattfand und globale Variablen sich möglicherweise geändert haben. Nach dem Wartezustand wird geprüft ob im Feld *THRSTAT_RECEIVED_SOFTINT* ein gültiger Software-Interrupt steht - andernfalls wird die Operation mit einer Fehlermeldung beendet. Unabhängig vom Rückgabewert werden die veränderten Deskriptoreinträge wieder auf ihre Grundeinstellungen zurückgesetzt.

13.2.2 Übermittlung eines Software-Interrupts

Erkennt eine Kernel-Eintrittsroutine eine Überwachung der Software-Interrupts, so gibt sie die Behandlung des Software-Interrupts an die Routine *kremote_received* (*remote.c, sched.h*) weiter. Diese Routine weckt den überwachenden Thread wieder auf und entfernt ihn ggf. mittels *ksched_del_timeout* (*timeout.c, sched.h*) aus der Timeout-Liste und löscht das Timeout-Flag *THRSTAT_TIMEOUT*. Anschließend friert sie den aktuellen Thread über *sysc_freeze_subject* ein.

13.2.3 Erkennung der Überwachung

Der wohl schwierigste Teil bei der Überwachung von Software-Interrupts ist die Erkennung der Überwachung und die korrekte Wiedergabe. Die Überwachung muss bei jeder Art von Software-Interrupt (Systemaufruf, leerer Software-Interrupt, Software-Interrupt eines IRQs / einer Exception) seperat implementiert werden und gestaltet sich unterschiedlich schwierig.

Erkennung bei leeren Software-Interrupts

Bei leeren Software-Interrupts ist die Erkennung am leichtesten. Hier gibt die Funktion *i386_handle_emptyint* (*intr.c, sched.h*) lediglich durch Aufruf von *kremote_received* (*remote.c, sched.h*) den Software-Interrupt weiter, wenn *THRSTAT_SOFTINT_LISTENER_SID* einen gültigen Wert hat.

Erkennung bei Systemaufrufen

Bei Systemaufrufen gestaltet sich die Sache insgesamt schon etwas schwerer. Hier muss die Erkennung für jeden Systemaufruf seperat implementiert sein, da ja jeder Systemaufruf einen eigenen Low-Level-Handler in der Datei *sysc.s* besitzt. In der Regel sieht der dafür zuständige Abschnitt wie folgt aus (am Beispiel von *alloc*):

```
#
# Redirect the system call if we are selected for
# recv_softints
#
pushl   %eax
pushl   %ebp
movl    current_t, %ebp
addl    $60, %ebp               # THRTAB_SOFTINT_LISTENER_SI
movl    (%ebp), %eax
cmpl    $0, %eax
```

```
            je      i386_sysc_alloc_pages_norm      # Normal system cal

            # Redirect it
            pushal
            pushl   $0xC0
            call    kremote_received
            addl    $4, %esp
            cmpl    $0, %eax
            popal
            jne     i386_sysc_alloc_pages_norm      # Normal execution,

            # Return to user mode
            popl    %ebp
            popl    %eax
            jmp     i386_do_context_switch
    i386_sysc_alloc_pages_norm:
```

Im ersten Teil des Erkennungscodes wird überprüft, ob *THRTAB_SOFTINT_LISTENER_SID* (das liegt am 60. Byte des Threadsdeskriptors) einen Wert ungleich 0 hat. Ist dem nicht so, wird der Systemaufruf normal bei *i386_sysc_alloc_pages_norm* (das Prefix *_norm* ist dabei bei allen Systemaufrufen identisch verwendet) fortgesetzt. Ist der Wert ungleich 0, wird *kremote_received* gestartet, um den Systemaufruf umzulenken. Nach dessen Aufruf wird geprüft, ob das System den Systemaufruf zusätzlich dennoch ausführen soll (*kremote_received* gibt einen Wert ungleich 0 zurück), oder ob der Systemaufruf zu ignorieren ist (siehe Flag RECV_TRACE_SYSCALL in 7.7.1) und nach dessen Ende mit dem Kontext-Wechsel durch *i386_do_context_switch* fortgefahren.

Erkennung eines Software-Interrupts einer Exceptions oder eines IRQs

Da die Software-Interrupts, die von Exceptions oder IRQs verwendet werden, prinzipiell auch umgeleitet werden können müssen (es könnte ja z.B. sein, dass ein emuliertes Betriebssystem die von den IRQs belegten Softwareinterrupts für seine Systemaufrufe verwendet), aber andererseits die Software-Interrupts von IRQs und Exceptions vor Zugriffen aus dem Benutzermodus gesperrt werden müssen, ist für die Erkennung dieser Software-Interrupts eine besondere Maßnahme erforderlich.

Immer wenn ein Thread einen Software-Interrupt aufruft, der von einem IRQ oder einer Exception belegt ist (abgesehen von einigen wenigen, zulässigen Exceptions), erzeugt er eine allgemeine Schutzverletzung. Im Fehlercode der Schutzverletzung ist erkennbar, dass die Fehlerursache ein ungültiger Zugriff auf die IDT war. Ebenfalls im Fehlercode enthalten ist die Nummer des aufgerufenen Software-Interrupts. Diese Nummer kann an *kremote_received* dann zur weiteren Verarbeitung weitergegeben werden.

Diese Überprüfung findet in der kernelseitigen Exceptionbehandlung *ksched_handle_except* (*intr.c*, *sched.h*) statt, wenn bei einem Thread die Interrupt-Umleitung aktiviert ist. Ein Problem ist jedoch, dass bei Auftreten der allgemeinen Schutzverletzung (Exception 13) der Programmzeiger an der Adresse der fehlerhaften Instruktion stehenbleibt. Der Kernel muss daher den Programmzeiger des Threads entsprechend ändern. Ein Interrupt kann bei x86-Systemen durch die Instruktionen *INT*, *INT3* un *INTO* ausgelöst werden. Da diese Instruktionen unterschiedliche Längen haben, bleibt dem Kernel nichts anderes übrig, als die Codestelle zu disassemblieren, die den Fehler auslöste, um dann dementsprechend den Programmzeiger zu ändern. An dieser Stelle muss der Kernel auf den virtuellen Benutzeradressraum zugreifen, was über eine Änderung des *ES*-Segmentselektos geschieht. Die komplette Codestelle lautet wie folgt:

```
    asm("pushl %%es\n"
```

```
        "movw $0x33, %%ax\n"
        "movw %%ax, %%es\n"
        "movb %%es:(%%ebx), %%al\n"
        "popl %%es\n"
        : "=a" (l__instr)
        : "b"(l__stack[12])
        : "memory"
       );
```

Zunächst wird der ES-Deskriptor auf den Stack gesichert. Anschließend wird der Selektor *0x33* geladen, der auf das Datensegment im Benutzermodus verweist. Dieses Datensegment wird nach ES geladen. Über das Register EBX, das im voraus mit dem Code-Zeiger des Programms geladen wurde, wird anschließend die Instruktion ausgelesen. Zum Schluß wird ES wiederhergestellt.

Weitergabe normaler Exceptions

Verursacht ein Thread eine Exception, so wird die Exception nicht an den Paging-Dämon, sondern an den überwachenden Thread weitergeleitet. Der überachende Thread kann die weiteren Daten über die aufgetretene Exception aus dem Deskriptor des fehlerhaften Threads auslesen.

Die Umleitung geschiet in den Routinen *kpaged_pagefault_to_usermode* und *kpaged_exception_to_usermode* (*paged.c*), die im Kapitel über den Paging-Dämon näher erläutert werden.

Kapitel 14

Pagingoperationen

14.1 Der Paging-Dämon

Der *hymk* besitzt zu einem Prozess im System eine besondere Beziehung: Dem Paging Dämon. Dieser Prozess hat erweiterte Rechte, aber auch Aufgaben. So kann der Paging-Dämon ungehindert auf den virtuellen Adressraum anderer Prozesse zugreifen und wird möglicherweise in späteren Versionen des *hymk* ebenfalls direkten Zugriff auf Seitentabellen und Seitenverzeichnisse anderer Prozesse erhalten.

Auf der anderen Seite ist der Paging Dämon für das saubere Beenden von Prozessen und Threads zuständig und nimmt alle unbehandelten Ausnahmefehler entgegen. Da er an dieser Stelle einen gewissen Flaschenhals darstellt, muss der Paging-Dämon sehr effizient implementiert sein.

14.1.1 Weitergabe von Ausnahmefehlern

Wie jedes andere Programm auch, kommuniziert der Paging-Dämon mit dem Kernel über den *sync*-Systemaufruf. Der Thread, der sich beim Kernel über den Aufruf *set_paged* als Paging-Dämon angemeldet hat, synchronisiert sich ständig mit dem Kernel-Platzhalter. Tritt eine Exception auf, synchronisiert der Kernel den fehlerhaften Thread mit dem Paging-Dämon und friert ihn anschließend ein. Der Paging-Dämon erhält als Rückgabewert der *sync*-Operation die SID des fehlerhaften Threads und kann über den Thread-Deskriptor die Informationen zum aufgetretenen Fehler auslesen.

Dieser Vorgang ist im wesentlichen in der Routine *kpaged_message_send* (*paged.c*) implementiert. Diese Routine wird durch die Exception-Behandlung des Kernels ausgelöst. Tritt eine Exception auf, startet der Kernel über *ksched_exception_to_user_mode* (*intr.c*) normalerweise die Funktion *kpaged_handle_exception* (*paged.c*, *sched.h*)[1]. Diese Funktion prüft zunächst ob ein Paging-Dämon installiert ist oder nicht und ob die Exception durch den Paging-Dämon ausgelöst wurde. Existiert kein Paging-Dämon oder verursachte der Paging-Dämon selbst die Exception, so wird das System angehalten.

In den anderen Fällen wird die Exception näher untersucht. Handelte es sich um einen Page Fault (*EXC_X86_PAGE_FAULT*), so wird die Funktion *kpaged_pagefault_to_usermode* (*paged.c*) mit der Untersuchung und Behandlung der Exception weiter beauftragt. Handelte es sich um eine andere Exception, so wird diese an die Funktion *kpaged_exception_to_usermode* (*paged.c*) weitergeleitet.

Gibt eine der Funktionen 1 als Rückgabewert zurück, so wird dieser Rückgabewert auch an den Aufrufer von *kpaged_handle_exception* weitergegeben. Der Rückgabewert 1 bedeutet, dass die Exception im Kernel-Modus verbleibt und der Kernel das System anhalten soll.

[1] Ausnahmen hiervon werden nur bei überwachten Interrupts, einer Kernel-Mode-Exception oder aber bei einer durch Copy-On-Write verursachten Exception gemacht.

Behandlung eines Page-Faults

Bei einem Page Fault wird zunächst von *kpaged_pagefault_to_usermode* die Adresse des Page-Faults durch auslesen des *CR2*-Registers der CPU ermittelt. Über den Aufruf *kpaged_send_pagefault* wird der Page-Fault dann an den Paging-Dämon weitergeleitet: Dazu wird der zuständige Seitendeskriptor mittels *kmem_get_table* ermittelt (*page.h*), wobei fehlende Seitentabellen weder erzeugt, noch nachgeladen werden. Die ermittelten Daten werden in die entsprechenden Felder des Deskriptors des betroffenen Threads eingetragen. Ist eine Umleitung auf die Software-Interrupts des Threads aktiviert, wird die Exception mit Hilfe von *kremote_received* (*remote.c, sched.h*) weiterdelegiert. Andernfalls wird sie über *kpaged_message_send* (*paged.c*) an den Paging-Dämon übergeben.

Die Felder mit den Informationen über den Page-Fault sind: *THRTAB_LAST_EXCPT_NUMBER* (plattformunabhängige Nummer der Exception - in diesem Fall *EXC_INVALID_PAGE*), *THRTAB_LAST_EXCPT* (x86-spezifische Nummer der Exception - in diesem Fall *EXC_X86_PAGEFAULT*), *THRTAB_LAST_EXCPT_ERR* (x86-spezifischer Fehlercode), *THRTAB_LAST_EXCPT_ADDRESS* (Programmadresse, an dem die Exception auftrat), *THRTAB_PAGEFAULT_LINEAR_ADDRESS* (Lineare Adresse an welcher der Page-Fault auftrat) und *THRTAB_PAGEFAULT_DESCRIPTOR* (Seitendeskriptor der betrofefnen Seite).

Hierbei sei anzumerken, dass ein Page-Fault, den der Aufruf *unmap* durch das gesetzte Flag *GENFLAG_PAGED_PROTECTED* ausgelöst hat, den Parameter *THRTAB_LAST_EXCPT_NR_PLATTFORM* auf den Wert 0xABCD setzt und weder eine IP, noch einen Fehlercode übermittelt. Tritt dieser speziell Fall auf, kann der Paging-Dämon daraus schließen, dass auf die betroffene Seite eine *unmap*-Operation ausgeführt werden soll.

Behandlung einer Exception

Alle anderen Exception werden durch *kapged_exception_to_usermode* weiterverarbeitet. Diese Funktion übersetzt zunächst die x86-spezifische Exceptionnummer in eine *hymk*-spezifische Fehlernummer. Anschließend werden die Deskriptor-Fehler des fehlerhaften Threads mit den Informationen über die Exception gefüllt. Diese Felder sind *THRTAB_LAST_EXCPT_NUMBER* (plattformunabhängige Nummer der Exception), *THRTAB_LAST_EXCPT_NR_PLATTFORM* (x86-spezifische Nummer der Exception), *THRTAB_LAST_EXCPT_ERROR_CODE* 86-spezifischer Fehlercode) und *THRTAB_LAST_EXCI* (Programmadresse, an dem die Exception auftrat).

Existiert eine Interrupt-Umleitung, wird diese über *kremote_received* (*remote.c, sched.h*) getätigt. Andernfalls wird die Exception über *kpaged_message_send* (*paged.c*) an den Paging-Dämon weitergegeben.

14.1.2 Installation des Paging-Dämons

Der Paging-Dämon wird durch den Systemaufruf *set_paged*, implementiert in der Funktion *sysc_set_paged* (*paged.c, sysc.h*) installiert. Hierbei wird die interne Variable *paged_thr_sid* auf die SID des aufrufenden Threads gesetzt. Diese SID wird später bei Synchronisationen mit dem Paging-Dämon verwendet. Im Prozess-Deskriptor des Paging-Dämons wird das Feld *PRCTAB_IS_PAGED* auf 1 gesetzt, wodurch alle Threads dieses Prozesses die volle Berechtigung des Paging-Dämons erhalten.

14.1.3 Anmerkungen

Derzeit ist noch kein Paging-Dämon für HydrixOS implementiert worden. Da es sich während der Implementierung des Dienstes möglicherweise ergeben kann, dass dem Paging-Dämon weitere Rechte eingeräumt werden müssen oder Schnittstellenänderungen erforderlich sein können, müssen die Angaben über die Schnittstelle zum Paging-Dämon derzeit noch als vorläufig betrachtet werden. Weitere

Details werden erst zur Verfügung stehen können, wenn die Entwicklung des Paging-Dämons angegangen wird.

14.2 Copy-On-Write

Der *hymk* bietet zur Beschleunigung von Kopierverfahren zwischen Prozessen eine Implementierung des unter UNIX üblichen Copy-On-Write-Verfahrens an. Bei diesem Verfahren werden beim Kopieren von Speicherseiten, diese nicht sofort kopiert, sondern lediglich zwischen den Prozessen gemeinsam genutzt. Die Speicherseiten werden als schreibgeschützt markiert und für das Copy-On-Write-Verfahren selektiert. Wird nun auf die Seite geschrieben, tritt ein Page-Fault auf. Der Kernel findet dabei heraus, dass die Seite für Copy-On-Write markiert ist und kopiert diese. Die Kopie wird in den Adressraum des anderen Prozesses beschreibbar und ohne Copy-On-Write-Flag eingeblendet und - sollte kein weiterer Prozess die Seite nutzen - im Quellprozess das Copy-On-Write gelöscht. Durch dieses Verfahren soll Rechenzeit und Speicherplatz gespart werden, da Erfahrungsgemäß oft Kopien von Daten zwischen Prozessen meistens lediglich gelesen und nur selten geändert werden (wie z.B. bei einer gemeinsam genutzten Programmbibliothek).

Implementierung des Verfahrens bei Ausnahmefehlern

Die Selektierung für das Copy-On-Write-Verfahren erfolgt durch den Systemaufruf *sysc_map*, wenn der Parameter *MAP_COPYONWRITE* gesetzt ist. Hierbei wird beim Einblenden der Seite in den Zieladressraum, wie auch im Quelladressraum das Copy-On-Write-Flag gesetzt und der Schreibschutz der Seite aktiviert.

Bei einem Zugriff auf eine Seite tritt auf x86-Plattformen die Page-Fault-Exception auf. Diese wird auf unterster Ebene durch *ksched_handle_except* behandelt. Erkennt die Routine einen Pagefault (*EXC_X86_PAGEFAULT*) und wurde dieser durch einen Zugriff auf eine schreibgeschützte Speicherseite im Benutzermodus erzeugt, so wird die weitere Überprüfung durch *kmem_copy_on_write* (*page.c*, *mem.h*) übernommen. Wird hier eine andere Ursache für die Exception erkannt, wird die Exception normal über *ksched_exception_to_user_mode* (*intr.c*) weiterbehandelt.

Die Funktion *kmem_copy_on_write* sucht anschließend die Adresse des Zugriffs und gibt die Durchführung des Copy-On-Write-Auftrags an *kmem_do_copy_on_write* (*page.c*, *mem.h*) weiter. *kmem_do_copy* prüft nun die Ursache, indem nach dem gesetzten Flag *GENFLAG_DO_COPYONWRITE* im Seitendeskriptor gesucht wird. Ist er nicht vorhanden, wird die Funktion mit einem entsprechenden Rückgabewert beendet. Anschließend wird geprüft, ob die Seite für Copy-On-Write noch in Frage kommt, oder ob der Prozess, der den Page-Fault auslöste nur noch der einzige Prozess ist, der auf die Seite zugreift. Ist dem so, wird die Copy-On-Write-Markierung und der Schreibschutz der Seite aufgehoben und der Page-Fault als behandelt erklärt.

Andernfalls wird durch *kmem_alloc_user_pagefrage* (*alloc.c*, *mem.h*) ein neuer Seitenrahmen reserviert. Im UMCA werden sowohl Quell- als auch Zielrahmen eingeblendet, so dass der Kernel Zugriff auf die Seiten erhalten kann (siehe 6.1). Anschließend wird der TLB für den verwendeten UMCA-Bereich invalidiert und die Daten werden zwischen der Quell- und Zielseite kopiert. Nachdem die Seiten aus dem UMCA wieder entfernt wurden, wird die neue Seite in den Adressraum des Aufrufers eingeblendet und der Aufrufer aus der Besitzerliste der Quellseite durch *kmem_free_user_pageframe* (*alloc.c*, *mem.h*) entfernt. Zum Schluß wird ebenfalls der TLB für die neue Seite invalidiert, so dass nach Wiedereintritt in den Benutzermodus der Zugriff auf die neue Seite erfolgen kann.

Gibt *kmem_do_copy_on_write* bzw. *kmem_copy_on_write* den Rückgabewert 0 zurück, gilt der Page-Fault als behandelt. Bei anderen Rückgabewerten muss die Exception, wie bereits erwähnt, weitergeleitet werden.

Implementierung des Verfahrens bei *map*

Während eines *map*-Vorgangs kann ebenfalls die Durchführung von Copy-On-Write erforderlich werden: Soll eine Seite gemeinsam zwischen Prozessen genutzt werden, die für *Copy-On-Write* vorgemerkt ist, dann wird das zwangsläufig zu Inkonsistenzen führen, da spätestens bei einem Schreibzugriff durch einen der beiden Prozesse der Copy-On-Write-Mechanismus anlaufen würde und die gemeinsame Speichernutzung aufgebrochen werden würde. Ein direktes Sharen einer mit Copy-On-Write markierten Speicherseite ist nur dann erwünscht, wenn das Share selbst ebenfalls ein Copy-On-Write-Share ist (map mit *MAP_COPYONWRITE*).

Folglich muss währernd der Etablierung des gemeinsamen Speicherbereichs die Seite im Quelladressraum zunächst mittels Copy-On-Write kopiert werden, so dass dann zwischen beiden Prozessen tatsächlich eine gemeinsam genutze Speicherseite existiert. Dazu ruft *sysc_map* die Funktion ebenfalls *kmem_do_copy_on_write* auf. Der Kopierforgang läuft dann wie bereits bekannt ab.