# Aufbau und Implementierung der HydrixOS-Basisbibliothek hyBaseLib

Friedrich Gräter

13. Februar 2007

**Lizenz**

**Vorwort**

An unterster Stelle des Benutzer-Raums von HydrixOS steht die sog. *hyBaseLib*. Diese Bibliothek bietet ganz grundlegende Funktionen für die Implementierung des höheren HydrixOS-Systems an. Dazu gehört die Bindings zum *hymk*, Primitiven zur Speicherverwaltung, Stringfunktionen oder eine einfache Threadverwaltung.

In diesem Dokument sollen die grundlegenden Prinzipien dieser *hyBaseLib* zunächst erläutert werden. Anschließend folgt eine Übersicht über die einzelnen API-Aufrufe der *hyBaseLib* und zum Schluss eine vollständige Dokumentation der Implementierung dieser Bibliothek.

# Inhaltsverzeichnis

# Teil I

# Konzepte der hyBaseLib

# Kapitel 1

# Gliederung der HydrixOS-API

Die HydrixOS-API gliedert sich in mehrere Ebenen und dementsprechend auch in mehrere Bibliotheken auf. An unterster Stelle steht die Grundlagenbibliothek *hyBaseLib* [1]. Diese Bibliothek stellt grundlegende Funktionen für den Systembetrieb zur Verfügung. Dazu gehören Bindings für die Systemaufrufe des *hymk*, dazu gehört die Verwaltung des *Thread Local Storage* eines Threads, dazu gehört die Verwaltung des virtuellen Adressraums, Primitiven zur Threadverwaltung und Synchronisation über Mutexe. Die *hyBaseLib* bildet somit die Grundlage des Benutzer-Raums unter HydrixOS. Alle weiteren API-Bibliotheken bauen auf dieser kleinen Bibliothek auf.

[1] Für die Namenskonvention von Bibliotheken gilt, das übliche Schema aus Prefix *hy*, dem Bibliotheksnamen und dem Suffix *Lib*. Bei Bibliotheken, die nur für eine bestimmte Programmiersprache entworfen worden sind, folgt dem Suffix *Lib* ein Kürzel, dass die Sprache kennzeichnet (z.B. *hyStdLibC* für die C-Standardbibliothek unter HydrixOS)

# Kapitel 2

# Konzepte und Aufgaben der *hyBaseLib*

## 2.1 Verwaltung des *thread local storage*

Der HydrixOS Mikrokernel *hymk* stellt jedem Thread einen kleinen, individuellen Speicherbereich zur Verfügung, in dem der Thread private Daten ablegen kann. Dieser Speicherbereich wird *thread local storage* oder *TLS* genannt. Ein solcher *TLS* ist sinnvoll, da ein Thread sehr oft auf Daten zugreifen müssen, die speziell diesem Thread zugeordnet sind und nur für diesen relevant sind. Dazu gehört z.B. der aktuelle Fehlerzustand der API-Bibliothek. Es wäre eine Verschwendung von CPU-Zeit, wenn jeder Thread zunächst eine Threadtabelle entlanglaufen müsste, um seine lokalen Variablen zu finden - daher ist eine Speicherseite des virtuellen Adressraums speziell für den *TLS* vorreserviert. Wird beim Prozesswechsel der aktuelle Thread ausgetauscht, so wird dementsprechend die Speicherseite des aktuellen *TLS* auf die des Threads geändert, der nun die CPU erhält. Dadurch erhält jeder Thread den Eindruck, als ob an einer bestimmten, festgelegten Speicherstelle im virtuellen Adressraum ein lokaler Speicherbereich liegt, der private Inhalte speichern kann.

Da der Speicherplatz in diesem TLS jedoch sehr begrenzt ist und es eine Reihe von Standard-Anwendungen für den TLS gibt (der besagte Fehlerzustand), scheint es sinnvoll, wenn der Speicherplatz auf dem TLS bereits auf unterster Ebene der API-Bibliothek verwaltet wird. Folglich ist die Verwaltung des TLS Aufgabe der *hyBaseLib*.

Die *hyBaseLib* teilt den TLS in zwei gleich große Bereiche: Die lokale Zone und die globale Zone. Die Struktur der lokalen Zone gilt nur für den aktuellen Thread - d.h. wird dort ein Speicherbereich reserviert, so ist diese nur im TLS des aktuellen Threads reserviert. Die Struktur der globalen Zone hingegen gilt für alle Threads. Wird in der globalen Zone ein Speicherbereich reserviert, so gilt er in allen TLS als reserviert. Dadurch können z.B. lokale Threadvariablen realisiert werden, deren Inhalt zwar an den Thread gebunden ist, deren Adresse jedoch für alle Threads gleich ist. Der Fehlerzustand des aktuellen Threads liegt z.B. immer an der gleichen Adresse im virtuellen Adressraum - unabhängig vom laufenden Thread.

Da der Speicher auf dem TLS sehr begrenzt ist, kann sein Speicher nur sehr rudimentär verwaltet werden. Daher ist es zwar möglich auf dem TLS einen Bereich zu reservieren - ist er aber einmal reserviert, so kann er nicht mehr freigegeben werden. Dies reicht für die meisten TLS-basierten Aufgaben aus, da diese meist Zeiger auf dem TLS ablegen, die für den gesamten Prozess- oder Threadbetrieb weiterbestehen sollen. Für eine komplexere TLS-Verwaltung sind High-Level-Funktionen nötig, die den TLS über eine externe Tabelle verwalten.

Zur Verwaltung der globalen Zone ist die Funktion *tls_global_alloc* zuständig, für die Verwaltung der lokalen Zone ist die Funktion *tls_local_alloc* zuständig. Beim Initialisieren der Grundlagenbibliothek wird der aktuelle Fehlerzustand global auf allen TLS alloziiert und ein Zeiger darauf in *tls_errno* abgelegt. Ein Thread der seinen privaten Fehlerzustand auslesen will, braucht also nur diesen globalen Zeiger zu dereferenzieren.

## 2.2 Bindings der Systemaufrufe

Damit ein Anwendungsprogramm überhaupt die Fähigkeiten des HydrixOS Mikrokernels nutzen kann, muss es in der Lage sein, dessen Systemaufrufe auszulösen. Da dies eine sehr plattformspezifische Tätigkeit ist, wird sie durch die Grundlagenbibliothek gekapselt. Die jeweiligen Systemaufrufe des *hymk* liegen als Funktion mit gleichem Namen, allerdings einem vorangehenden Prefix *hymk*, vor. Diese Funktionen nehmen ebenfalls den zurückgegebenen Fehlerwert des *hymk* entgegen und speichern ihn in den lokalen Fehlerzustand des Threads.

## 2.3 Kapselung plattformspezifischer Kernel-Schnittstellen

Einige Schnittstellen des *hymk* sind sehr plattformspezifisch Entworfen worden, um unnötigen Overhead zu vermeiden. Zu diesen Schnittstellen gehören u.a. die Hauptinfopage, die Prozess- und Threadtabelle, sowie die Aufrufe *hymk_alloc_pages*, *hymk_io_alloc*, *hymk_map*, *hymk_unmap* und *hymk_move*. Die HydrixOS-API enthält Aufrufe, die diese Schnittstellen plattformübergreifend nutzbar machen können.

#### Zugriff auf Infopages

Die Hauptinfopage kann durch den Aufruf *hysys_info_read*, die Prozesstabelle über den Aufruf *hysys_proctab_read* und die Threadtabelle über den Aufruf *hysys_thrtab_read* ausgelesen werden. Diese Aufrufe kapseln zwar nicht die Struktur dieser Schnittstellen (hierfür sind jedoch Makros in der Include-Datei *hymk/sysinfo.h* enthalten), aber sie kapseln den generellen Zugriff auf diese Info-Pages.

#### Zugriff auf Speicherverwaltungsaufrufe

Alle Speicherverwaltungsaufrufe des *hymk* (*hymk_alloc_pages*, *hymk_io_alloc hymk_map*, *hymk_unmap* und *hymk_move*) können nur eine implementierungsspezifische Anzahl von Speicherseiten auf einmal verarbeiten (für die derzeitige x86-Version des *hymk* sind es 8 MiB), da zu lange Arbeitszeiten im (derzeit nicht preemptiblen) Kernel-Modus zu vermeiden sind. Um dem Entwickler jedoch das ständige Zerlegen der Speicheroperationen in die implementierungsspezifischen Mengen zu erlassen, wurden die API-Aufrufe *hysys_alloc_pages*, *hysys_io_alloc*, *hysys_map*, *hysys_unmap* und *hysys_move* geschaffen, die diese Aufgaben automatisch erledigen.

## 2.4 Speicherzugriff

Da die Standard-Bibliotheken der meisten Programmiersprachen Funktionen für das Dateisystem benötigen, implementiert die Grundlagenbibliothek einige Aufrufe zur Verarbeitung von Speicherbereichen und Zeichenfolgen. Normalerweise ist es sinnvoll, wenn die Funktionen der jeweiligen Sprachen verwendet werden - die Aufrufe der *hyBaseLib* sind nur für Bibliotheken und Programme gedacht, die ohne Standardbibliothek ihrer Sprache auskommen müssen.

Zur Verarbeitung von Speicherbereichen mit fester Länge existieren die Funktionen *buf_copy* (Kopieren eines Puffers), *buf_fill* (Füllen eines Puffers mit einem Datenwort), *buf_compare* (Vergleich zweier Puffer), *buf_find_uint8*, *buf_find_uint16*, *buf_find_uint32*, *buf_find_uint64* (Finden eines Datenwortes bestimmter Länge in einem Puffer) und *buf_find_buf* (Finden einer größeren Datenfolge in einem Puffer). Zur Verarbeitung von nullterminierten UTF-8 und ASCII-Zeichenfolgen existieren die Funktionen *str_copy* (Kopieren eines String), *str_compare* (Vergleichen zweier Strings), *str_len* (Ermitteln der Länge eines Strings), *str_char* (Finden eines ASCII-Zeichens in einem String) und *str_find* (Finden eines Strings in einem String).

## 2.5 Organisation des virtuellen Adressraums

### 2.5.1 Regionen

Der virtuelle Adressraum wird durch die *hyBaseLib* auf oberster Ebene in Regionen unterteilt. Eine Region ist ein Bereich des virtuellen Adressraums, der einen einheitlichen Zweck erfüllt, eine bestimmte Speicherverwaltungsroutine benutzt und bestimmte feste Eigenschaften trägt. Ferner bestehen Regionen normalerweise über den gesamten Programmbetrieb oder zumindest sehr lange Zeiträume hinweg. Regionen können manuell über den Aufruf *reg_create* erstellt und über den Aufruf *reg_destroy* zerstört werden. Es existiert eine verkettete Liste *regions*, die alle Header einer Speicherregion miteinander verbindet. Der Header einer Speicherregion liegt normalerweise in deren ersten Speicherseite, eine Abweicung ist aber prinzipiell möglich. Sie sollte aber nur dann verwendet werden, wenn es unbedingt erforderlich ist (z.B. bei den Init-Modulen). Diese Speicherseite ist zumindest lesbar und beschreibbar.

Der Header einer Speicherregion speichert folgende Eigenschaften einer Region:

- Name der Region (32-Zeichen UTF-8)
- Identifizierungsnummer der Region (Integer-Zahl)
- Zugriffsrechte (*lesen, schreiben, ausführen, gemeinsame Nutzung, statische Inhalte*)
- Routinen für Speicherallokation und Freigabe
- Gesamtzahl gültiger Speicherseiten
- Anzahl lesbarer Speicherseiten
- Anzahl beschreibbarer Speicherseiten
- Anzahl ausführbarer Speicherseiten
- Anzahl gemeinsam genutzter Speicherseiten

Die Zugriffsrechte einer Speicherregion haben keinen direkten Einfluss auf die Zugriffsrechte der in Speicherseiten dieser Region. Sie dienen nur zur allgemeinen Information und sollten die Zugriffsrechte des Großteils aller Speicherseiten darstellen. Die Umsetzung dieser Zugriffsrechte sollte durch die Routine zur Speicherallokation stattfinden. Die Daten zur Anzahl von Speicherseiten müssen ebenfalls nicht korrekt sein und dienen viel mehr statistischen Zwecken. Sie sollten allerdings von der verwendeten Routine zur Speicherallokation aktualisiert werden.

Eine Region muss nicht zwingend mit gültigen Speicherseiten vollständig aufgefüllt sein. Sie muss auch nicht zwingend eine Allokations- und Freigabefunktion anbieten, sondern kann auch statische Inhalte haben.

### 2.5.2 Vordefinierte Regionen

Die *hyBaseLib* gibt einige Regionen standardmäßig[1] vor, die nach der Initialisierung der Bibliothek bestehen:

- Die *Code*-Region, welche den ausführbaren Maschinencode des Programms enthält. Diese Region besitzt keinen Allokator und darf nur ausgelesen und ausgeführt werden.

[1] In den ersten HydrixOS-Versionen werden die Code und die Daten-Region aus Ermangelung einer Software zum Laden von Programmen noch nicht aktiviert sein. Der Init-Prozess muss diese Regionen selbst initialisieren.

- Die *Daten*-Region, welche die statischen Daten des Programms enthält. Diese Region besitzt ebenfalls keinen Allokator und darf nur ausgelesen und beschrieben werden.
- Die *Stack*-Region, welche die Stacks der verschiedenen Threads enthält. Diese Region besitzt einen Allokator, darf aber nur ausgelesen und beschrieben werden[2].
- Die *Heap*-Region, welche den Heap (d.h. dynamisch alloziierte Daten) des Prozesses enthält. Diese Region besitzt ebenfalls einen Allokator und darf auch nur ausgelesen und beschrieben werden.
- Die *Mapping*-Region, welche zur Allokation von Page-Mappings speicher bereitstellt. Diese Region besitzt ebenfalls einen Allokator.

Diese vordefinierten Regionen haben Plattformabhängige Größen und Positionen im virtuellen Adressraum[3].

### 2.5.3 Dynamische Regionen

Prinzipiell können während des Betriebs weitere Regionen im verbleibenden virtuellen Adressraums während des Betriebs erstellt werden. Hierfür ist der Aufruf *reg_create* zuständig, der eine Region in einem gegebenen Speicherbereich anlegt. Mit dem Aufruf *reg_destroy* kann eine bestehende Region wieder vernichtet werden.

In der Regel ist das Anlegen von Regionen jedoch nicht nötig und auch kaum möglich, da meist der gesamte Adressraum mit Regionen belegt ist. Es ist daher meist sinnvoller Speicher im Map-Bereich der Mapping-Region zu verwenden.

## 2.6 Verwaltung des Heaps

### 2.6.1 Grobe Organisation des Heaps

Um nicht unnötig Seitenrahmen zu verschwenden, ist die Heap-Region normalerweise nicht vollständig mit Seitenrahmen aufgefüllt. Zunächst einmal ist der Bereich mit nutzbaren Seitenrahmen auf dem Heap begrenzt. Dieser Bereich (der sog. Nutzbereich des Heaps) fängt nach dem Regionsdeskriptor des Heaps an und geht bis zu einer festgelegten Obergrenze. Diese Obergrenze kann durch den Aufruf *mem_heap_inc* erweitert und mit *mem_heap_dec* herabgesenkt werden, wobei entsprechend mit Hilfe von *hymk_alloc_pages* neue Seitenrahmen dem Adressraum hinzugefügt, bzw. mit *hymk_unmap* entfernt werden. Liegen in der Mitte dieses Bereichs freie Speicherbereiche vor, die sich über mehre Speicherseiten hinweg erstrecken, so werden deren Seitenrahmen ebenfalls freigegeben und erst wieder bei Bedarf alloziiert.

### 2.6.2 API-Aufrufe zur Speicherverwaltung

Der Nutzbereich des Heaps selbst wird nun selbst in sog. Speicherblöcke zerteilt, welche die dynamischen erzeugten Daten des Prozeses enthalten können. Diese Speicherblöcke können durch den Aufruf von *mem_alloc* alloziiert und mit dem Aufruf *mem_free* wieder freigegeben werden. Der Aufruf *mem_realloc* kann Speicherblöcke vergrößern oder verkleinern, wobei dies oft ein Verschieben des Speicherblocks samt Inhalt bedeutet.

[2] In der derzeitigen Implementierung der HydrixOS-API bleibt die Stack-Region ungenutzt, da Stack-Bereiche über den Heap alloziiert werden.

[3] Die x86-Architektur hält Code, Stack und Daten im ersten GiB des Adressraums. Der Heap erhält das 2. GiB des virtuellen Adressraums. Das 3. GiB wird normalerweise durch das Mapping-Region verbraucht.

### 2.6.3 Beschreibung der Blockverwaltung

Jeder Speicherblock besitzt einen Header, der dem vom Programm genutzten Bereich des Speicherblocks vorausgeht. Dieser Header enthält Informationen über die Größe und Zustand des Blocks. Alle Blöcke - ob sie nun freie Speicherberiche oder belegte Speicherbereiche enthalten, sind grundsätzlich in einer großen verketteten Liste zusammengefasst, der sog *General Block List* (GBL). Alle Blöcke, die einen belgten Speicherbereich enthalten, sind ebenfalls in einer verketteten Liste zusammengefasst, der sog. *Used Block List* (UBL).

Die freien Speicherblöcke jedoch werden in mehreren unterschiedilchen Listen, den sog. *Free Block Lists* (FBL) verwaltet. Jede FBL hat eine Ordnungszahl (dies wird mit *FBL*(*n*) notiert, wobei *n* die Ordnungszahl ist), wobei die Anzahl der FBLs plattformabhängig ist. Eine FBL der Ordnungszahl n enthält alle freien Speicherbereiche, die im Größenbereich von $2^n$ bis $2^{n+1}$-1 liegen. Die x86-Architektur verwendet 31 getrennte Listen, wodurch prinzipiell freie Speicherbereiche bis zu einer Größe von bis zu 4 GiB verwaltet werden könnten. Bei der Größenbemessung wird der Header nicht mit eingeschlossen - d.h. ein Speicherblock muss zumindest so groß wie sein Header sein.

### 2.6.4 Der Allokationsalgorithmus

Die Allokation eines Speicherbereichs läuft wie folgt ab: Zunächst wird eine FBL ausgewählt, welche die Größenanforderung so exakt wie möglich erfüllen kann. Enthält diese keine Speicherblöcke, wird die nächst größere FBL ausgewählt. Die gewählte FBL wird nun nach einem Block durchsucht, der exakt der Speicheranforderung entspricht.

Wird kein exakt passender freier Speicherblock gefunden, so muss ein Block zerteilt werden. Bei dieser Zerteilung entsteht der neue Nutzblock und ein Verschnitt, der seinerseits als freier Speicherblock registriert wird. Damit hier eine erneute Suche nicht nötig wird, wird bereits bei der Suche nach dem exakt passenden Speicherblock jeweils ein Block notiert, bei dem ein sehr kleiner Verschnitt entsteht (sog. *best-fit*)[4]. War die Suche in der aktuellen FBL erfolglos, so wird in der nächst höheren FBL weitergesucht. Wird in keiner FBL ein passender Block gefunden, so wird der Heap durch den Aufruf von *mem_heap_inc* entsprechend vergrößert und am Ende des Heaps ein neuer Block angelegt. Um das Heap-Ende schneller auffinden zu können, wird der Heap-Deskriptor mit der höchsten Speicheradresse im Voraus gespeichert.

Dieses *quick-best-fit* Verfahren versucht zum einen die lange Suchzeit eines üblichen *best-fit* zu reduzieren, da die Suche zuerst mit den Blockgrößen anfängt, die am wahrscheinlichsten dem *best-fit*-Prinzip entsprechen und alle viel zu kleinen Blockgrößen grundsätzlich übergeht.

### 2.6.5 Der Freigabealgorithmus

Da die GBL alle Speicherblöcke enthält, kann bei der Freigabe eines Speicherblocks rasch festgestellt werden, ob ein Nachbarblock des freizugebenden Blocks selbst ein freier Speicherblock ist und somit eine Vereinigung der Blöcke zu einem großen Block möglich ist. Liegt ein Speicherbereich am oberen Ende des Heaps, so wird nach einer evtl. Vereinigung mit Nachbarblöcken der Heap durch Aufruf von *mem_heap_dec* verkleinert und der Block völlig aus der Tabelle entfernt oder zumindest verkleinert. Entsteht bei der Verkleinerung ein Bereich, der kleiner als ein Blockheader ist, so wird dieser Bereich dem benachbarten Block gutgeschrieben.

Liegt ein Bereich mitten im Heap, so wird der freie Block am Anfang der jeweils passenden FBL eingefügt. Erstreckt sich ein Block über mehr als eine Seitengrenzen hinweg, so werden Seiten, die

[4] Es entsteht kein Verschnitt, wenn der theoretische Verschnitt kleiner als ein Block-Header ist. In diesem Fall, wird der neue Bereich einfach entsprechend größer alloziiert. Restbereiche die kleiner als ein Block-Header sind, könnten sonst nicht einmal ihren eigenen Block-Header enthalten.

innerhalb des freien Blocks liegen freigegeben und dies im Blockheader vermerkt, so dass bei der Allokation diese Seitenrahmen wiederhergestellt werden können.

### 2.6.6 Größenänderung bei Blöcken

Soll ein Block vergrößert werden, wird zunächst überprüft, ob anschließend an den aktuellen Block ein freier Bereich (oder das Heap-Ende) liegt, mit dem die Vergrößerungsanforderung erledigt werden kann. Fehlt ein solcher Block, so wird ein neuer Bereich reserviert und der Blockinhalt dorthin kopiert.

Bei einer Verkleinerung bleibt der Block an gleicher Stelle bestehen und es wird lediglich am Blockende ein neuer, freier Block angelegt. Verkleinerungen die kleiner als die Größe eines Blockheaders sind, bleiben jedoch ohne Effekt.

## 2.7 Verwaltung der Mapping-Region

Der Algorithmus zur Verwaltung der Mapping-Speicherregion arbeitet im wesentlichen genauso, wie der Algorithmus zur Verwaltung des Heaps (siehe 2.6). Er unterscheidet sich jedoch darin, dass nur Blöcke alloziiert werden können, deren Größe ein ganzahliges Vielfaches einer Speicherseite beträgt und die am Anfang einer Speicherseite beginnen. Ferner liegen die Blockdeskriptoren dieser Region nicht als Header vor dem alloziierten Speicherblock, sondern werden von den Speicherblöcken getrennt im Heap abgelegt. Ebenfalls sind die Look-Up-Tables für der Mapping-Region auf dem Heap abgelegt.

Zur Allokation eines Speicherblocks in dieser Region ist der Aufruf *pmap_alloc* zuständig. Dieser Aufruf reserviert einen Speicherbereich in der Mapping-Region - allerdings ohne diesen Speicherbereich mit physischen Seitenrahmen zu unterlegen. Dies ist sinnvoll, da viele Mapping-Speicherbereiche üblicherweise mit gemeinsam genutzten Seiten aufgefüllt werden. Soll ein Speicherbereich jedoch mit Unterlegung physischer Seitenrahmen reserviert werden, so kann dies durch den Aufruf *pmap_mapalloc* gemacht werden. Der Aufruf *pmap_free* gibt einen Bereich in der Mapping-Region frei und entfernt dabei automatisch alle in diesem Bereich liegenden Seitenrahmen.

## 2.8 Verwaltung des Stacks

Die *hyBaseLib* erlaubt die Verwaltung von Stacks mit unterschiedlicher Größe. Die Bibliotheksroutine zur Reservierung von Stack-Bereichen trägt den Namen *mem_stack_alloc*. Die Freigaberoutine trägt den Namen *mem_stack_free*.

Derzeit werden alle Stacks auf dem Heap eines Prozesses gespeichert. In späteren Versionen ist eine Verwaltung der Stacks innerhalb der Stack-Region erforderlich.

## 2.9 Einfache Thread-Verwaltung

Die *hyBaseLib* bietet einige rudimentäre Funktionen zur Verwaltung von Threads an. Diese Funktionen tragen das Prefix *blthr_* (BaseLib Threads). Es gibt dabei Funktionen zum Verdrängen des aktuellen Threads (*blthr_yield*), zum Anhalten (*blthr_freeze*) und Aufwecken (*blthr_awake*) von Threads, zum Erstellen neuer Threads (*blthr_create*), zum Beenden von Threads (*blthr_kill*), zum Festlegen von Terminierungsfunktionen des Thread (*blthr_atexit*), zur Selbstterminierung eines Threads (*blthr_finish*) und zum Aufräumen bestehender Threads-Datenstrukturen nach Beendigung von Threads (*blthr_cleanup*). Es existiert intern der Aufruf *blthr_init*, welcher die interne Start-Up-Funktion des Threads für die *hyBaseLib* enthält.

Jeder Thread wird dabei durch seinen Thread-Deskriptor identifiziert. Der Thread-Deskriptor ist auf dem Heap abgelegt. Auf dem TLS des betroffenen Threads liegt ein Zeiger auf die Adresse seines Deskriptors. Der Deskriptor enthält die Startadresse des verwendeten Stack-Bereichs, sowie die Liste der Funktionen, die bei Beendigung des Threads ausgeführt werden sollen.

Auf dem TLS legt die *hyBaseLib* eine Variable *tls_my_thread* ab, welche einen Zeiger auf die Datenstruktur des aktuellen Threads enthält. Die Thread-Datenstrukturen selbst verfügen über einen Mutex, der zur Synchronisation des Zugriffs auf den Deskriptor zwischen Erzeuger des Threads und dem Thread selbst gedacht ist. Sollten mehrere Threads gleichzeitig Zugriff auf einen Thread-Deskriptor haben, so muss die Synchronisation dieses Zugriffs extern geregelt sein. Dies ist sinnvoll, da in den meisten Situationen eben nur der Thread selbst oder sein Erzeuger auf den Deskriptor zugreifen und der Mehrfachzugriff von mehreren Seiten her eine komplexere und weniger performante Synchronisation erfordern würde.

## 2.10 Mutex-Synchronisation

Für die Synchronisation von Threads stellt die *hyBaseLib* Mutexe zur Verfügung. Ein Mutex dient zum Schutz einer kritischen Sektion, die nur von einem einzigen Thread gleichzeitig betreten werden darf (z.B. zum Schutz einer gemeinsam genutzten Speichervariable). Dies wird wie folgt erreicht: Ein Mutex ist eine Variable, die nur durch atomare Operationen verarbeitet werden darf. Sie kann den Zustand frei (0) und gesperrt (1) einnehmen. Damit ein Thread die kritische Sektion betreten kann, muss er zunächst den Mutex sperren. Dabei wird atomar geprüft, ob der Mutex bereits gesperrt ist oder nicht. Ist er noch nicht gesperrt, so wird er im gleichen Zug atomar gesperrt. Ist er bereits gesperrt, so gibt der Thread seine CPU-Zeit mittels *blthr_yield* ab und versucht die Variable erneut zu sperren, wenn er die CPU erhält. Ist der Mutex gesperrt, so darf der Thread die kritische Sektion betreten und die gewünschte Operation ausführen. Nachdem die Operation ausgeführt wurde, muss er den Mutex wieder atomar freigeben, so dass andere Threads die Sektion wieder betreten können.

Zum Sperren eines Mutex dient die Operation *mtx_trylock*, die lediglich versucht, den Mutex zu sperren, ohne den Thread die CPU-Zeit abgeben zu lassen. Ferner gibt es die Operation *mtx_lock*, die bei gescheitertem Sperrungsversuch zur Abgabe der CPU-Zeit führt. Zum Schutz des Threads kann für *mtx_lock* ein Timeout definiert werden. Ferner gibt es den Aufruf *mtx_unlock*, der zur Entsperrung eines gesperrten Mutex führt.

Um eine zu einseitige Nutzung des Mutexes zu vermeiden, verfügt der Mutex über ein Latency-Bit. Versucht ein anderer Thread einen bereits gesperrten Mutex zu sperren, so kann er das Latency-Bit setzen. Ist es gesetzt, so wird der entsperrende Thread nach dem Aufruf von *mtx_unlock* eine Verdrängung mittels *blthr_yield* auslösen, so dass der andere Thread die Möglichkeit erhält, den Mutex zu sperren.

Ein Mutex muss durch das Makro *MTX_NEW* vor der ersten Verwendung initialisiert werden. Für globale Variablen muss während der Variablendefinition das Makro *MTX_DEFINE* verwendet werden.

## 2.11 XML-Parsing

Das HydrixOS-Dateisystem bietet viele Meta-Informationen in Form von XML-Daten an. Es ist somit häufig erforderlich, dass selbst einfache Programme bereits XML-Daten parsen können. Hierfür bietet die hyBaseLib einen minimalen XML-Parser an. Dieser Parser unterstützt nicht alle Ausdrucksmöglichkeiten der XML-Sprache. Unterstützt werden normale Tags mit Inhalt (z.B. <info> </info) und Tags ohne Inhalt (<info/>). Tags mit Inhalt werden nach Textdaten oder untergeordneten Tags durchsucht. Kommentare (<!– comment –>), Whitespaces und Processing-Tags (<? process data ?>) werden vollständig ignoriert. DTD-Informationen (<! info >) sollten derzeit nicht enthalten sein, da

der Parser sie als normale Tags auffasst. XML-Attribute werden auch nicht unterstützt - alle Daten müssen sich zwischen Tags befinden. CDATA-Bereiche (<element [foo]>) werden ebenfalls nicht unterstützt. Es steht aber offen, ob weitere Fähigkeiten der XML-Sprache später hinzugefügt werden.

Der Parser ist primär für die Analyse der Dateisystem-Metadaten gedacht. Damit dies möglichst unkompliziert geschehen kann, wandelt die Funktion *spxml_create_tree* einen String mit XML-Daten in einen Baum um, dessen Kind-Elemente jeweils XML-Unterelemente (entweder Daten, leere Tags oder Tags mit weiteren Unterelementen) sind. Jedes Element enthält außerdem einen Zeiger auf den Bereiche des Eingabestrings, der den Namen des Elements definiert hat - d.h. solange der XML-Baum verwendet wird, darf auch der Eingabestring nicht gelöscht werden. Ein bestimmtes Element in einem Baum kann mit Hilfe eines XPath-ähnlichen Pfades über die Funktion *spxml_resolve_pfad* gefunden werden. Wird der Baum nicht mehr verwendet, kann sein Speicher mittels *spxml_destroy_tree* vernichtet werden.

Die Entity-Referenzen < > & " und ' können außerdem mit Hilfe des Aufrufs *spxml_replace_stdentities* ersetzt werden.

# Teil II

# Schnittstellen der HyBaseLib

# Kapitel 3

# Übersicht über die Schnittstelle

## 3.1 Aufrufe zur Verwaltung des *thread local storage* (TLS)

### 3.1.1 *tls_global_alloc* - TLS-Bereich global alloziieren

**Deklaration:**

```
#include <hydrixos/tls.h>
void** tls_global_alloc(void);
```

**Eingabeparameter:**

Keine.

**Rückgabewerte:**

Zeiger auf den TLS-Bereich (der selbst ein void* ist).

**Beschreibung:**

Alloziiert einen TLS-Bereich, der auf dem TLS jedes Threads an gleicher Position liegen soll.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_NOT_ENOUGH_MEMORY` | Der globale TLS-Speicher ist aufgebraucht. |

### 3.1.2 *tls_local_alloc* - TLS-Bereich lokal alloziieren

**Deklaration:**

```
#include <hydrixos/tls.h>
void** tls_local_alloc(void);
```

**Eingabeparameter:**

Keine.

**Rückgabewerte:**

Zeiger auf den TLS-Bereich (der selbst ein void* ist)

**Beschreibung:**

Alloziiert einen TLS-Bereich, der nur auf dem TLS des aktuellen Threads sichtbar sein soll.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| ERR_NOT_ENOUGH_MEMORY | Der lokale TLS-Speicher ist aufgebraucht. |

## 3.2 Implementierung der Kernelaufrufe

### 3.2.1 *hymk_alloc_pages* - Speicherseiten alloziieren

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_alloc_pages(void* adr, unsigned pages);
```

**Eingabeparameter:**

| | |
|---|---|
| adr | Startadresse des Mapping |
| pages | Anzahl der zu reservierenden Speicherseiten |

**Rückgabewerte:**

Anzahl der gemappten Pages

**Beschreibung:**

Reserviert *pages* freie Seitenrahmen und fügt sie an der festgelegten Startadresse *adr* im aktuellen virtuellen Adressraum ein. Enthält ein Teil des Zielbereichs im aktuellen Adressraum bereits Seitenrahmen, so werden diese nicht überschrieben (entsprechend wird die Anzahl der gemappten Pages im Rückgabewert re. Der Zielbereich darf weder ganz, noch teilweise im Kernel-Adressraum liegen. Er darf ebenfalls nicht an Adresse 0 und auch nicht im Bereich des *thread local storage* liegen.

**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, an Adresse 0 oder im *thread local storage*. |

### 3.2.2 *hymk_create_thread* - Neuen Thread erstellen

**Deklaration:**

```
#include <hydrixos/hymk.h>
sid_t hymk_create_thread(void* ip, void* sp)
```

**Eingabeparameter:**

| | |
|---|---|
| ip | Startadresse des neuen Threads |
| sp | Anfangsadresse des Stacks des neuen Threads |

**Rückgabewerte:**

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. |

### 3.2.3 *hymk_create_process* - Neuen Prozess erstellen

**Deklaration:**

```
#include <hydrixos/hymk.h>
sid_t hymk_create_process(void* ip, void* sp);
```

**Eingabeparameter:**

| | |
|---|---|
| ip | Startadresse des neuen Threads |
| sp | Anfangsadresse des Stacks des neuen Threads |

**Rückgabewerte:**

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. |

### 3.2.4 *hymk_set_controller* - Controller-Thread festlegen

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_set_controller(sid_t ctl);
```

**Eingabeparameter:**

| | |
|---|---|
| ctl | SID des neuen Controller-Threads |

**Rückgabewerte:**

Keiner.

**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. |

### 3.2.5 *hymk_destroy_subject* - Subjekt zerstören

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_destroy_subject(sid_t subj);
```

**Eingabeparameter:**

| | |
|---|---|
| subj | SID des betroffenen Prozess- oder Thread-Subjekts |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Vernichtet ein Subjekt *subj* und gibt seine Datenstrukturen frei. Die Operation kann grundsätzlich nur von Threads des Paging-Dämons ausgeführt werden.

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.

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-Subject 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 ebenfalls der Prozess danach aufgelöst (siehe oben).

Ein laufender Thread des Paging-Dämons kann sich nicht selbst beenden, sondern muss durch einen anderen Thread des Paging-Dämons beendet werden. Dadurch wird unnötige Komplexität im Kernel vermieden und kann gleichzeitig eine vernünftige Freigabe des Thread-Stacks im Usermode durch den Paging-Dämon realisiert werden.

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:**

Nur Paging-Dämon.

**Fehlercodes:**

| | |
|---|---|
| `ERR_PAGING_DAEMON` | Nur der Paging-Dämon darf diesen Systemaufruf tätigen. |
| `ERR_INVALID_SID` | Die verwendete SID ist ungültig. |

### 3.2.6 *hymk_chg_root* - Root-Rechte eines Prozesses ändern

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_chg_root(sid_t subj, int mode);
```

**Eingabeparameter:**

| | | |
|---|---|---|
| subj | SID des Prozess-Subjekt, das seinen root-Modus ändern soll. | |
| mode | Subjekt | |
| | 0 | soll den Root-Modus verlassen. |
| | 1 | soll den Root-Modus betreten. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Wechselt einen Prozess *subj* 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. |

### 3.2.7 *hymk_freeze_subject* - Subjekt einfrieren.

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_freeze_subject(sid_t subj);
```

**Eingabeparameter:**

| | |
|---|---|
| subj | SID des Threadsubjekts, das eingefroren werden soll. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Friert einen Thread *subj* 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 *hymk_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. |

### 3.2.8 *hymk_awake_subject* - Weckt ein Subjekt wieder auf

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_awake_subject(sid_t subj);
```

**Eingabeparameter:**

| | |
|---|---|
| subj | SID des Threadsubjekt, das aufgeweckt werden soll. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Weckt einen eingefrorenen Thread *subj* 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. |

### 3.2.9 *hymk_yield_thread* - CPU-Zeit abgeben

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_yield_thread(sid_t recv);
```

**Eingabeparameter:**

| | |
|---|---|
| recv | SID des Threads, der den Rest der effektiven Priorität empfangen soll |

**Rückgabewerte:**

Keiner.

**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 *recv* 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.

### 3.2.10 *hymk_set_priority* - Threadpriorität ändern

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_set_priority(sid_t subj, int prior, int cls);
```

**Eingabeparameter:**

| | |
|---|---|
| subj | SID des betroffenen Threads |
| prior | Neue Priorität (Zahl von 0 bis 40) |
| class | Neue Schedulingklasse |
| | 0 SCHED_REGULAR |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Ändert die statische Priorität eines Threads *subj* auf *prior*, sowie dessen Scheduling-Klasse auf *cls* (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. |

### 3.2.11 *hymk_allow* - Speicheroperation zulassen

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_allow(sid_t subj, sid_t me, void* adr, unsigned pages, int op);
```

**Eingabeparameter:**

| | |
|---|---|
| subj | 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 gesperrt. *kernel* hat keine Wirkung. Die SID muss nicht gültig sein. |
| me | 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. |
| adr | Startadresse des für die Operation genehmigten Bereichs |
| pages | Anzahl der Seiten des Bereichs |
| op | 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. |

**Rückgabewerte:**

Keiner.

**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. |

### 3.2.12 *hymk_map* - Speicherseiten mappen

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_map(sid_t subj, void* adr, unsigned pages, int flags, uintptr_t dest_offset);
```

**Eingabeparameter:**

| | | | |
|---|---|---|---|
| subj | SID des Threads auf den die Operation angewandt werden soll | | |
| adr | Startadresse des Quellbereichs im eigenen Adressraum | | |
| pages | Anzahl der Seiten des Bereichs | | |
| flags | 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! |
| dest_offset | Offset im Zielbereich | | |

**Rückgabewerte:**

Keiner.

**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 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).

Ist eine Seite im Quellmapping für Copy-On-Write markiert, so wird diese Eigenschaft an den Zieladressraum weitervererbt - d.h. es entsteht kein gemeinsamer Speicherbereich, sondern lediglich eine Kopie des Bereichs. Hier sollten Maßnahmen im Benutzermodus verwendet

werden (z.B. die Verwendung „frisch“ alloziierter Speicherseiten für Mappings), um etwaige Probleme mit dem Copy-On-Write-Mechanismus zu vermeiden.

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).

Die möglichen Zugriffsrechte auf eine Speicherseite können nicht angehoben werden - d.h. ist eine Seite nur lesbar, kann sich nicht weider beschreibbar gemacht werden.

**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_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. |

### 3.2.13 *hymk_unmap* - Mapping reduzieren/aufheben

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_unmap(sid_t subj, void* adr, unsigned pages, int flags);
```

**Eingabeparameter:**

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

adr — Startadresse des für die Operation genehmigten Zielbereichs

pages — Anzahl der Seiten des Bereichs

flags — 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. |
| `UNMAP_AVAILABLE` | 1 | Die Speicherseiten sollen als nicht-vorhanden markiert werden. Der physikalische Seitenrahmen wird freigegeben, wenn kein anderer Prozess ihn mehr benutzt. |
| `UNMAP_WRITE` | 2 | Die Seiten dürfen nur noch ausgelesen werden. |
| `UNMAP_EXECUTE` | 3 | Die Speicherseiten dürfen nicht mehr ausgeführt werden. (auf x86 ineffektiv) |

**Rückgabewerte:**

Keiner.

**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_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. |

### 3.2.14 *hymk_move* - Speicherseiten zwischen Prozessen verschieben

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_move(sid_t subj, void* adr, unsigned pages, int flags, uintptr_t dest_offset);
```

**Eingabeparameter:**

| | | | |
|---|---|---|---|
| subj | SID des Threads auf den die Operation angewandt werden soll | | |
| adr | Startadresse des Quellbereichs im lokalen Adressraum | | |
| pages | Anzahl der Seiten des Bereichs | | |
| flags | Flags: | | |
| | `MAP_READ` | 1 | Die Seiten können ausgelesen werden. |
| | `MAP_READWRITE` | 2 | Die Seiten können beschrieben werden (auf x86 impliziert dies MAP_READ). |
| | `MAP_EXECUTABLE` | 4 | Die Speicherseiten können ausgeführt werden (auf x86 nicht verfügbar). |
| dest_offset | Offset im Zielbereich | | |

**Rückgabewerte:**

Keiner.

**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. |

### 3.2.15 *hymk_sync* - Threads synchronisieren

**Deklaration:**

```
#include <hydrixos/hymk.h>
sid_t hymk_sync(sid_t subj, long tm, int resync);
```

**Eingabeparameter:**

| | |
|---|---|
| subj | SID der Gegenseite |
| tm | Länge des Timeouts (ungefähr) in Millisekunden (0 = kein Warten, 0xFFFFFFFF = unendlich) |
| resync | Anzahl der Resync-Vorgänge (0 = kein Resync) |

**Rückgabewerte:**

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. |

### 3.2.16 *hymk_io_allow* - I/O-Zugriffsrechte freischalten

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_io_allow(sid_t subj, int flags);
```

**Eingabeparameter:**

| | | | |
|---|---|---|---|
| subj | SID des Zielprozesses | | |
| flags | 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:**

Keiner.

**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. |

### 3.2.17 *hymk_io_alloc* - I/O-Speicherbereich alloziieren

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_io_alloc(uintptr_t phys, void* adr, unsigned pages, int flags);
```

**Eingabeparameter:**

| | | | |
|---|---|---|---|
| phys | Zu reservierender Quellspeicherbereich | | |
| adr | Zielspeicherbereich des Mappings | | |
| pages | Anzahl der zu mappenden Seiten | | |
| flags | 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:**

Keiner.

**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. |

### 3.2.18 *hymk_recv_irq* - IRQ überwachen

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_recv_irq(int irqn);
```

**Eingabeparameter:**

| | |
|---|---|
| irqn | Zu überwachender IRQ |

**Rückgabewerte:**

Keiner.

**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. |

### 3.2.19 *hymk_recv_softints* - Software-Interrupts abgreifen

**Deklaration:**

```
#include <hydrixos/hymk.h>
int hymk_recv_softints(sid_t subj, long tm, int flags);
```

**Eingabeparameter:**

| | |
|---|---|
| subj | Zu überwachender Thread |
| tm | Timeout, (ungefähr) in Millisekunden (0 = kein Timeout, 0xFFFFFFFF unendliches Timeout) |
| flags | Flags |

| | | |
|---|---|---|
| RECV_AWAKE_OTHER | 1 | Der zu beobachtende Thread soll beim Start der Beobachtung mit *awake_subject* aufgeweckt werden. |

**Rückgabewerte:**

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_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. |

### 3.2.20 *hymk_read_regs* - Register auslesen

**Deklaration:**

```
#include <hydrixos/hymk.h>
reg_t hymk_read_regs(sid_t subj, int tp);
typedef struct {
    int regs[SYSTEM_REGREAD_COUNT];
}reg_t;
/* For x86-Plattforms: */
#define SYSTEM_REGREAD_COUNT 4
```

**Eingabeparameter:**

| | | | |
|---|---|---|---|
| subj | SID des Threads, dessen Register gelesen werden sollen | | |
| tp | 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:**

| | |
|---|---|
| regs[0] | (1) |
| regs[1] | (2) |
| regs[2] | (3) |
| regs[3] | (4) |

**Beschreibung:**

Liest einige Register eines Threads aus. Hierbei ist anzumerken, dass das Binding dieses Aufrufs plattformabhängig ist.

**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. |

### 3.2.21 *hymk_write_regs* - Register schreiben

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_write_regs(sid_t subj, int tp, reg_t rv);
typedef struct {
    int regs[SYSTEM_REGREAD_COUNT];
}reg_t;
/* For x86-Plattforms: */
#define SYSTEM_REGREAD_COUNT 4
```

**Eingabeparameter:**

| | | | |
|---|---|---|---|
| subj | SID des Threads, in dessen Register geschrieben werden sollen | | |
| tp | Zu schreibender Register-Typus | | |
| | `REGS_X86_GENERIC` | 0 | (1) EAX<br>(2) EBX<br>(3) ECX<br>(4) EDX |
| | `REGS_X86_INDEX` | 1 | (1) ESI<br>(2) EDI<br>(3) EBP |
| | `REGS_X86_POINTERS` | 2 | (1) ESP<br>(2) EIP |
| | `REGS_X86_EFLAGS` | 3 | (1) EFLAGS<br>Nur Änderung des ersten Bytes möglich |
| regs | Zu schreibende Register (rv.regs[0] - rv.regs[3] werden auf (1) - (4) abgebildet). | | |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Ändert die Register eines Threads. Hierbei ist anzumerken, dass das Binding dieses Aufruf stark plattformabhängig ist.

**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. |

### 3.2.22 *hymk_set_paged* - Prozess als Paging-Dämon erklären

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hymk_set_paged(void);
```

**Eingabeparameter:**

Keiner.

**Rückgabewerte:**

Keiner.

**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. |

### 3.2.23 hymk_test_page

**Deklaration:**

```
#include <hydrixos/hymk.h>
unsigned hymk_test_page(uintptr_t adr, sid_t sid);
```

**Eingabeparameter:**

| | |
|---|---|
| adr | Adresse der Speicherseite |
| sid | Prozess SID des Zieladressraums (*invalid* oder *null* für den aktuellen Adressraum) |

**Rückgabewerte:**

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. |

## 3.3 Kapselung plattformspezifischer Kernel-Schnittstellen

### 3.3.1 *hysys_alloc_pages* - Speicherseiten alloziieren

**Deklaration:**

```
#include <hydrixos/system.h>
void hysys_alloc_pages(void* adr, unsigned pages);
```

**Eingabeparameter:**

| | |
|---|---|
| adr | Startadresse des Mapping |
| pages | Anzahl der zu reservierenden Speicherseiten |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Reserviert *pages* freie Seitenrahmen und fügt sie an der festgelegten Startadresse *adr* 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 an Adresse 0 und auch nicht im Bereich des *thread local storage* liegen.

**Beschränkungen:**

Dieser Aufruf hat keine Einschränkungen bei der Anzahl von Speicherseiten, die auf einen Aufruf gleichzeitig verarbeitet werden können. Er kapselt dies durch mehrfaches Aufrufen von *hymk_alloc_pages.*

**Fehlercodes:**

| | |
|---|---|
| `ERR_NOT_ENOUGH_MEMORY` | Es steht nicht genug Speicher zur Verfügung. |
| `ERR_INVALID_ADDRESS` | Der Zielbereich lag ganz oder teilweise im Kernel-Adressraum, an Adresse 0 oder im *thread local storage*. |

### 3.3.2 *hysys_io_alloc* - I/O-Speicherbereich alloziieren

**Deklaration:**

```
#include <hydrixos/system.h>
void hysys_io_alloc(uintptr_t phys, void* adr, unsigned pages, int flags);
```

**Eingabeparameter:**

| | |
|---|---|
| phys | Zu reservierender Quellspeicherbereich |
| adr | Zielspeicherbereich des Mappings |
| pages | Anzahl der zu mappenden Seiten |
| flags | 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:**

Keiner.

**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:**

Dieser Aufruf hat keine Einschränkungen bei der Anzahl von Speicherseiten, die auf einen Aufruf gleichzeitig verarbeitet werden können. Er kapselt dies durch mehrfaches Aufrufen von *hymk_io_alloc.*

**Fehlercodes:**

| | |
|---|---|
| ERR_NOT_ROOT | Die Operationsbeschränkungen wurden übertreten. |
| ERR_INVALID_ADDRESS | Der verwendete Ziel- oder Quellbereich ist ungültig. |
| ERR_INVALID_ARGUMENT | Die verwendeten Operationsflags sind ungültig. |

### 3.3.3 *hysys_map* - Speicherseiten mappen

**Deklaration:**

```
#include <hydrixos/hymk.h>
void hysys_map(sid_t subj, void* adr, unsigned pages, int flags, uintptr_t dest_offset);
```

**Eingabeparameter:**

subj — SID des Threads auf den die Operation angewandt werden soll

adr — Startadresse des Quellbereichs im eigenen Adressraum

pages — Anzahl der Seiten des Bereichs

flags — 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! |

dest_offset — Offset im Zielbereich

**Rückgabewerte:**

Keiner.

**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 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).

Ist eine Seite im Quellmapping für Copy-On-Write markiert, so wird diese Eigenschaft an den Zieladressraum weitervererbt - d.h. es entsteht kein gemeinsamer Speicherbereich, sondern lediglich eine Kopie des Bereichs. Hier sollten Maßnahmen im Benutzermodus verwendet

werden (z.B. die Verwendung „frisch“ alloziierter Speicherseiten für Mappings), um etwaige Probleme mit dem Copy-On-Write-Mechanismus zu vermeiden.

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).

Die möglichen Zugriffsrechte auf eine Speicherseite können nicht angehoben werden - d.h. ist eine Seite nur lesbar, kann sich nicht weider beschreibbar gemacht werden.

**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.

Dieser Aufruf hat keine Einschränkungen bei der Anzahl von Speicherseiten, die auf einen Aufruf gleichzeitig verarbeitet werden können. Er kapselt dies durch mehrfaches Aufrufen von *hymk_map*.

**Fehlercodes:**

| | |
|---|---|
| `ERR_ACCESS_DENIED` | Die Operationsbeschränkungen wurden übertreten. |
| `ERR_INVALID_SID` | Die verwendete Ziel-SID ist ungültig. |
| `ERR_INVALID_ARGUMENT` | Die verwendeten Operationsflags sind ungültig. |
| `ERR_INVALID_ADDRESS` | Die Quelladresse überschneidet sich mit dem Kernel-Adressraum. |

### 3.3.4 *hysys_unmap* - Mapping reduzieren/aufheben

**Deklaration:**

```
#include <hydrixos/system.h>
void hysys_unmap(sid_t subj, void* adr, unsigned pages, int flags);
```

**Eingabeparameter:**

| | |
|---|---|
| subj | SID des Threads auf den die Operation angewandt werden soll. Wird *null* oder *invalid* übergeben, wird als Ziel der aktuelle Thread verwendet. |
| adr | Startadresse des für die Operation genehmigten Zielbereichs |
| pages | Anzahl der Seiten des Bereichs |
| flags | 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. |
| `UNMAP_AVAILABLE` | 1 | Die Speicherseiten sollen als nicht-vorhanden markiert werden. Der physikalische Seitenrahmen wird freigegeben, wenn kein anderer Prozess ihn mehr benutzt. |
| `UNMAP_WRITE` | 2 | Die Seiten dürfen nur noch ausgelesen werden. |
| `UNMAP_EXECUTE` | 3 | Die Speicherseiten dürfen nicht mehr ausgeführt werden. (auf x86 ineffektiv) |

**Rückgabewerte:**

Keiner.

**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.

Dieser Aufruf hat keine Einschränkungen bei der Anzahl von Speicherseiten, die auf einen Aufruf gleichzeitig verarbeitet werden können. Er kapselt dies durch mehrfaches Aufrufen von *hymk_unmap*.

**Fehlercodes:**

| | |
|---|---|
| `ERR_ACCESS_DENIED` | Die Operationsbeschränkungen wurden übertreten. |
| `ERR_INVALID_SID` | Die verwendete Ziel-SID ist ungültig. |
| `ERR_INVALID_ARGUMENT` | Die verwendeten Operationsflags sind ungültig oder die Zieladresse überschneidet sich mit dem Kernel-Adressraum. |

### 3.3.5 *hysys_move* - Speicherseiten zwischen Prozessen verschieben

**Deklaration:**

```
#include <hydrixos/system.h>
void hysys_move(sid_t subj, void* adr, unsigned pages, int flags, uintptr_t dest_offset);
```

**Eingabeparameter:**

| | | | |
|---|---|---|---|
| subj | SID des Threads auf den die Operation angewandt werden soll | | |
| adr | Startadresse des Quellbereichs im lokalen Adressraum | | |
| pages | Anzahl der Seiten des Bereichs | | |
| flags | Flags: | | |
| | `MAP_READ` | 1 | Die Seiten können ausgelesen werden. |
| | `MAP_READWRITE` | 2 | Die Seiten können beschrieben werden (auf x86 impliziert dies MAP_READ). |
| | `MAP_EXECUTABLE` | 4 | Die Speicherseiten können ausgeführt werden (auf x86 nicht verfügbar). |
| dest_offset | Offset im Zielbereich | | |

**Rückgabewerte:**

Keiner.

**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.

Dieser Aufruf hat keine Einschränkungen bei der Anzahl von Speicherseiten, die auf einen Aufruf gleichzeitig verarbeitet werden können. Er kapselt dies durch mehrfaches Aufrufen von *hymk_move.*

**Fehlercodes:**

| | |
|---|---|
| `ERR_ACCESS_DENIED` | Die Operationsbeschränkungen wurden übertreten. |
| `ERR_INVALID_SID` | Die verwendete Ziel-SID ist ungültig. |
| `ERR_INVALID_ARGUMENT` | Die verwendeten Operationsflags sind ungültig oder die Zieladresse überschneidet sich mit dem Kernel-Adressraum. |

### 3.3.6 *hysys_info_read* - Hauptinfopage auslesen

**Deklaration:**

```
#include <hydrixos/system.h>
uint32_t hysys_info_read(unsigned int num);
```

**Eingabeparameter:**

| | |
|---|---|
| num | Nummer des Eintrags in der Hauptinfopage (*MAININFO_*-Konstanten in *hymk*/*sysinfo.h* ) |

**Rückgabewerte:**

Gelesener Wert aus der Hauptinfopage.

**Beschreibung:**

Liest aus der Hauptinfopage den Eintrag mit Nummer *num* aus und gibt dessen Inhalt zurück.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültige Infopage-Nummer. |

### 3.3.7 *hysys_proctab_read* - Prozesstabelle auslesen

**Deklaration:**

```
#include <hydrixos/system.h>
uint32_t hysys_proctab_read(sid_t sid, unsigned int num);
```

**Eingabeparameter:**

| | |
|---|---|
| sid | SID des Prozess-Subjekts. |
| num | Nummer des Eintrags in der Hauptinfopage (*PRCTAB_*-Konstanten in *hymk/sysinfo.h* ) |

**Rückgabewerte:**

Gelesener Wert aus der Prozesstabelle.

**Beschreibung:**

Liest aus dem Prozesstabelleneintrag für den Prozess *sid* das Element *num* aus.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültige Infopage-Nummer. |
| `ERR_INVALID_SID` | Ungültige Prozess-SID. |

### 3.3.8 *hysys_thrtab_read* - Threadtabelle auslesen

**Deklaration:**

```
#include <hydrixos/system.h>
uint32_t hysys_thrtab_read(sid_t sid, unsigned int num);
```

**Eingabeparameter:**

| | |
|---|---|
| sid | SID des Thread-Subjekts. |
| num | Nummer des Eintrags in der Hauptinfopage (*PRCTAB_*-Konstanten in *hymk/sysinfo.h* ) |

**Rückgabewerte:**

Gelesener Wert aus der Threadtabelle.

**Beschreibung:**

Liest aus dem Threadtabelleneintrag für den Thread *sid* das Element *num* aus.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültige Infopage-Nummer. |
| `ERR_INVALID_SID` | Ungültige Thread-SID. |

## 3.4 Speicherzugriff

### 3.4.1 *buf_copy* - Kopieren von Speicherbereichen

**Deklaration:**

```
#include <hydrixos/stdfun.h>
size_t buf_copy(void* dest, const void* src, size_t num);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich. |
| src | Zeiger auf den Quellbereich. |
| num | Anzahl der zu kopierenden Bytes. |

**Rückgabewerte:**

Anzahl der kopierten Bytes.

**Beschreibung:**

Kopiert einen Speicherbereich *src* von *num* Bytes nach *dest*.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Ziel- oder Quellzeiger ist ungültig (z.B. NULL). |

### 3.4.2 *buf_fill* - Speicherbereich füllen

**Deklaration:**

```
#include <hydrixos/stdfun.h>
size_t buf_fill(void* dest, size_t long num, uint8_t val);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich. |
| num | Anzahl der zu füllenden Bytes |
| val | Zu schreibender Wert |

**Rückgabewerte:**

Anzahl der geschriebenen Bytes.

**Beschreibung:**

Füllt einen Wert *val* in den Zielspeicherbereich *dest*, bis *num* Bytes geschrieben worden sind.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Zielzeiger ist ungültig (z.B. NULL). |

### 3.4.3 *buf_compare* - Speicherbereich vergleichen

**Deklaration:**

```
#include <hydrixos/stdfun.h>
int buf_compare(const void* dest, const void* src, size_t num);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich. |
| src | Zeiger auf den Quellbereich |
| num | Größe des Quellbereichs |

**Rückgabewerte:**

Gibt an, ob der Zielbereich größer (> 0), kleiner (< 0) oder gleich (== 0) gegenüber dem Quellbereich ist.

**Beschreibung:**

Vergleicht zwei Speicherbereiche *dest* und *src* binär miteinander. Hierbei wird entweder solange vergleichen, bis ein unterschiedlicher Wert entdeckt wurde (die Differenz der beiden Werte wird zurückgegeben) oder die Grenze *num* erreicht wurde.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Ziel- oder Quellzeiger ist ungültig (z.B. NULL). |

### 3.4.4 *buf_find_uintN* - Speicherbereich suchen

**Deklaration:**

```
#include <hydrixos/stdfun.h>
void* buf_find_uint8(const void* dest, size_t num, uint8_t val);
void* buf_find_uint16(const void* dest, size_t num, uint16_t val);
void* buf_find_uint32(const void* dest, size_t num, uint32_t val);
void* buf_find_uint64(const void* dest, size_t num, uint64_t val);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich. |
| num | Größe des Bereichs in Byte |
| val | Zu suchendes Wort |

**Rückgabewerte:**

Zeiger auf das erste Vorkommen.

NULL bei Fehlschlag.

**Beschreibung:**

Sucht ein Datenwort *val* in einem Zielbereich *dest*, der die Größe *num* hat. Dabei muss *num* ein Vielfaches von *sizeof*(val) sein. Das Datenwort *val* ist dabei je nach Funktion vom Typ *uint8_t*, *uint16_t*, *uint32_t* oder *uint64_t*.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Zielzeiger ist ungültig (z.B. NULL). |
| `ERR_INVALID_ARGUMENT` | Das Argument *num* ist kein Vielfaches von *sizeof*(*val*). |

### 3.4.5 *buf_find_buf* - Speicherbereich suchen

**Deklaration:**

```
#include <hydrixos/stdfun.h>
void* buf_find_buf(const void* dest, size_t dnum, const void *src, size_t snum);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich. |
| dnum | Größe des Zielbereichs in Byte |
| src | Zu suchende Bytefolge |
| snum | Größe der Bytefolge |

**Rückgabewerte:**

Zeiger auf das erste Vorkommen.

NULL bei Fehlschlag.

**Beschreibung:**

Sucht eine Bytefolge *src* mit der Größe *snum* in einem Zielbereich *dest*, der die Größe *dnum* hat.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Ziel- oder Quellzeiger ist ungültig (z.B. NULL). |
| `ERR_INVALID_ARGUMENT` | *snum* ist 0 oder größer als *dnum*. |

### 3.4.6 *str_copy* - Nullterminierte Zeichenfolge kopieren

**Deklaration:**

```
#include <hydrixos/stdfun.h>
size_t str_copy(utf8_t* dest, const utf8_t* src, size_t max);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich. |
| src | Zeiger auf den Quellbereich |
| max | Maximale Länge der Quellfolge |

**Rückgabewerte:**

| | |
|---|---|
| $\geq 0$ | Anzahl der tatsächlich kopierten Zeichen. |
| -1 | Es wurde bis *max* kein Terminierungszeichen gefunden |

**Beschreibung:**

Kopiert eine nullterminierte Zeichenfolge aus dem Puffer *src* in den Puffer *dest*. Diese Zeichenfolge hat höchstens die Länge *max*. Nach *max* Zeichen, oder wenn das Terminierungszeichen 0 gefunden wurde, wird das Kopieren abgebrochen. Das Terminierungszeichen wird ebenfalls kopiert. Enthält der String innerhalb von *max* kein Terminierungszeichen, so wird der Zielpuffer ebenfalls nicth mit 0 terminiert.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Ziel- oder Quellzeiger ist ungültig (z.B. NULL). |

### 3.4.7 *str_compare* - Nullterminierte Zeichenfolge vergleichen

**Deklaration:**

```
#include <hydrixos/stdfun.h>
int str_compare(const utf8_t* dest, const utf8_t* src, size_t max);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich. |
| src | Zeiger auf den Quellbereich |
| max | Maximale Länge der Quellfolge |

**Rückgabewerte:**

Der Zielbereich ist größer (>0), kleiner (<0) oder gleich (==0) dem Quellbereich.

**Beschreibung:**

Kopiert eine nullterminierte Zeichenfolge aus dem Puffer *src* in den Puffer *dest*. Diese Zeichenfolge hat höchstens die Länge *max*. Nach *max* Zeichen, oder wenn das Terminierungszeichen 0 gefunden wurde, wird das Kopieren abgebrochen. Das Terminierungszeichen wird ebenfalls kopiert.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Ziel- oder Quellzeiger ist ungültig (z.B. NULL). |

### 3.4.8 *str_len* - Länge einer nullterminierten Zeichenfolge ermitteln

**Deklaration:**

```
#include <hydrixos/stdfun.h>
size_t str_len(const utf8_t* dest, size_t max);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich |
| max | Maximale Länge der Zielfolge |

**Rückgabewerte:**

| | |
|---|---|
| $\geq 0$ | Länge der nullterminierten Zeichenfolge in Byte (ohne Terminierungszeichen). |
| -1 | Zeichenfolge ist innerhalb von *max* nicht nullterminiert. |

**Beschreibung:**

Gibt die Anzahl von Bytes in einer nullterminierten Zeichenfolge *src* ohne das Terminierungszeichen zurück.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Zielzeiger ist ungültig (z.B. NULL). |

### 3.4.9 *str_char* - Byte suchen

**Deklaration:**

```
#include <hydrixos/stdfun.h>
utf8_t* str_char(const utf8_t* dest, char sgn, size_t max);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich |
| sgn | Gesuchtes Byte |
| max | Maximallänge des Bereichs |

**Rückgabewerte:**

Zeiger auf das erste Vorkommen des gesuchten Bytes.

NULL, wenn das gesuchte Byte nicht vorgekommen ist.

**Beschreibung:**

Gibt einen Zeiger auf das erste Vorkommen des ASCII-Zeichens *sgn* in der Zeichenkette *dest*, die eine Maximallänge von *max* Byte hat, zurück. Diese Funktion ist nicht zur Suche von UTF-8-Zeichen geeignet!

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Zielzeiger ist ungültig (z.B. NULL). |

### 3.4.10 *str_find* - Zeichenfolge suchen

**Deklaration:**

```
#include <hydrixos/stdfun.h>
utf8_t* str_find(const utf8_t* dest, size_t dmax, const utf8_t* src, size_t smax);
```

**Eingabeparameter:**

| | |
|---|---|
| dest | Zeiger auf den Zielbereich |
| dmax | Maximalgröße des Zielbereichs |
| src | Zeiger auf den Quellbereich |
| smax | Maximalgröße des Quellbereichs |

**Rückgabewerte:**

Zeiger auf das erste Vorkommen der gesuchten Zeichenfolge.

NULL, wenn das gesuchte Zeichen nicht vorgekommen ist.

**Beschreibung:**

Gibt einen Zeiger auf das erste Vorkommen der nullterminierten Zeichenfolge *src* in der nullterminierten Zeichenkette *dest* zurück (wobei das Terminierungszeichen von *src* natürlich ignoriert wird). Die Zeichenfolgen haben jeweils eine Maximallänge von *smax* (für *src*) und *dmax* (für *dest*).

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Der Ziel- oder Quellzeiger ist ungültig (z.B. NULL). |

## 3.5 Verwaltung von Speicherregionen

### 3.5.1 *reg_create* - Region erzeugen

**Deklaration:**

```
#include <hydrixos/mem.h>
region_t* reg_create(region_t region);
typedef struct {
    char        name[32];         /* Name der Region */
    unsigned    id;               /* ID der Region */
    void*       start;            /* Startadresse */
    unsigned    pages;            /* Größe der Region in Seiten */
    int         flags;            /* Flags (siehe REGFLAGS_*) */
    unsigned    usable_pages;     /* Anzahl der brauchbaren Seiten */
    unsigned    readable_pages;   /* Anzahl der lesbaren Seiten */
    unsigned    writeable_pages;  /* Anzahl der beschreibbaren Seiten */
    unsigned    executable_pages; /* Anzahl der ausführbaren Seiten */
    unsigned    shared_pages;     /* Anzahl der gemeinsam genutzen Seiten */
    void* (*alloc)(size_t sz);    /* Allokator-Funktion (siehe mem_alloc) */
    void (*free)(void* ptr);      /* Freigabe-Funktion (siehe mem_free) */

    unsigned    chksum;           /* Prüfsumme über den Deskriptor */
    list_t      ls;               /* Elemente der verketteten Liste */
}region_t;

#define REGFLAGS_READABLE         1
#define REGFLAGS_WRITEABLE        2
#define REGFLAGS_EXECUTABLE       4
#define REGFLAGS_SHARED           8
```

**Eingabeparameter:**

region Regionsdeskriptor

**Rückgabewerte:**

Zeiger auf den Regionsdeskriptor.

NULL bei Fehler.

**Beschreibung:**

Erstellt eine Region anhand des übergebenen Regionsdeskriptors. Dabei wird an der angegebenen Adresse eine neue Speicherseite in den Adressraum eingeblendet, an welcher der Header der Region plaziert wird. Die Region wird in die verkettete Liste der Regionen eingehängt. Der Aufruf initialisiert automatisch die Deskriptorprüfsumme, welche über Startadresse und Seitenzahl der Region berechnet wird.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| ERR_INVALID_ADDRESS | Die Region liegt an einer ungültigen Adresse. |
| ERR_INVALID_ARGUMENT | Die Regionsflags sind ungültig. |
| ERR_RESSOURCE_BUSY | Die Region liegt in einer bereits bestehenden Region oder überlappt mit dieser. |

### 3.5.2 *reg_destroy* - Region vernichten

**Deklaration:**

```
#include <hydrixos/mem.h>
void reg_destroy(region_t *region);
```

**Eingabeparameter:**

region Zeiger auf den Regionsdeskriptor.

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Vernichtet eine bestehende Region. Dabei werden grundsätzlich alle Speicherseiten innerhalb der Region aus dem Adressraum automatisch entfernt.

**Beschränkungen:**

Keine.

**Fehlercodes:**

`ERR_INVALID_ARGUMENT` Der Regionsdeskriptor ist ungültig.

## 3.6 Speicherverwaltung (Hintergrundfunktionen)

### 3.6.1 *mem_heap_inc* - Heap vergrößern

**Deklaration:**

```
#include <hydrixos/mem.h>
void mem_heap_inc(unsigned pages);
```

**Eingabeparameter:**

| | |
|---|---|
| pages | Anzahl der Seiten, die dem Heap hinzugefügt werden sollen. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Fügt dem Heap *pages* neue physische Speicherseiten am oberen Ende hinzu.

**Beschränkungen:**

Keine.

**Anmerkung:**

Dieser Aufruf kann nicht zu einer sicheren Allokation von Heap-Speicher herangezogen werden, da er nicht die Heap-Verwaltung entsprechend auf den Speichergewinn einstellt. Viel mehr kann er aber dazu führen, dass die Heap-Verwaltung durcheinander gerät. Er sollte daher nur dann verwendet werden, wenn dem Aufrufer wirklich klar ist, was er dabei macht.

**Fehlercodes:**

| | |
|---|---|
| `ERR_NOT_ENOUGH_MEMORY` | Nicht genügend physischer Arbeitsspeicher verfügbar oder Heapgrenze erreicht. |
| `ERR_INVALID_ARGUMENT` | Die Anzahl der Seiten ist ungültig. |

### 3.6.2 *mem_heap_dec* - Heap

**Deklaration:**

```
#include <hydrixos/mem.h>
void mem_heap_dec(unsigned pages);
```

**Eingabeparameter:**

pages Anzahl der Seiten, die aus dem Heap entfernt werden sollen.

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Gibt *pages* Seiten vom oberen Ende des Heaps wieder frei.

**Beschränkungen:**

Keine.

**Anmerkung:**

Dieser Aufruf kann nicht zu einer sicheren Freigabe von Heap-Speicher herangezogen werden, da er nicht die Heap-Verwaltung entsprechend auf den Speicherverlust einstellt. Viel mehr kann er aber dazu führen, dass die Heap-Verwaltung durcheinander gerät. Er sollte daher nur dann verwendet werden, wenn dem Aufrufer wirklich klar ist, was er dabei macht.

**Fehlercodes:**

`ERR_INVALID_ARGUMENT` Die Anzahl von Speicherseiten ist ungültig.

### 3.6.3 *mem_stack_alloc* - Stack erzeugen

**Deklaration:**

```
#include <hydrixos/mem.h>
void* mem_stack_alloc(size_t sz);
```

**Eingabeparameter:**

| | |
|---|---|
| sz | Größe des zu reservierenden Stackbereichs |

**Rückgabewerte:**

Zeiger auf den Anfang neu erstellten Stack (nicht der Stackzeiger!).

NULL bei Fehlschlag.

**Beschreibung:**

Alloziiert einen Stack in der Stackregion und füllt ihn mit Speicherseiten auf.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_NOT_ENOUGH_MEMORY` | Die Stackregion enthält keinen Speicherplatz mehr oder es stehen nicht genügen physische Seitenrahmen zur Verfügung. |

### 3.6.4 *mem_stack_free* - Region vernichten

**Deklaration:**

```
#include <hydrixos/mem.h>
void mem_stack_free(void* stack);
```

**Eingabeparameter:**

| | |
|---|---|
| stack | Zeiger auf die Stackregion (nicht der Stack-Zeiger!) |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Gibt eine Stackregion wieder frei.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Die Stackregion ist ungültig. |

## 3.7 Allgemeine Speicherverwaltung (Heap)

### 3.7.1 *mem_alloc* - Speicher auf dem Heap reservieren

**Deklaration:**

```
#include <hydrixos/mem.h>
void* mem_alloc(size_t sz);
```

**Eingabeparameter:**

| | |
|---|---|
| sz | Anzahl der Byte. |

**Rückgabewerte:**

Zeiger auf den reservierten Bereich.

**Beschreibung:**

Reserviert *sz* Bytes auf dem Heap und gibt einen Zeiger darauf zurück.

NULL bei Fehlschlag.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_NOT_ENOUGH_MEMORY` | Es steht nicht genügend physischer Speicher oder Platz auf dem Heap zur Verfügung. |
| `ERR_INVALID_ARGUMENT` | Die Größenangabe ist ungültig. |
| `ERR_MEMORY_CORRUPTED` | Die Speicherverwaltung wurde beschädigt. Dies weist normalerweise auf einen Programmier- oder Hardwarefehler hin. Es wird empfohlen das Programm sofort zu beenden. |

### 3.7.2 *mem_realloc* - Größe eines Speicherbereichs ändern

**Deklaration:**

```
#include <hydrixos/mem.h>
void* mem_realloc(void* mem, size_t nsz);
```

**Eingabeparameter:**

| | |
|---|---|
| mem | Zeiger auf den bestehenden Speicherbereich. |
| nsz | Größe des neuen Speicherbereichs. |

**Rückgabewerte:**

Zeiger auf den neuen Speicherbereich.

NULL bei Fehlschlag.

**Beschreibung:**

Ändert die Größe eines Speicherbereichs *mem* auf *nsz*. Dabei kann der Bereich vergrößert oder verkleinert werden. Unter Umständen muss der Speicherbereich mit Inhalt verschoben werden, daher ist die Speicheradresse aus dem Rückgabewert als neue Adresse des Speicherbereichs zu verwenden. Bei einer Verkleinerung des Speicherbereichs find ein Datenverlust statt.

Wird *NULL* als Speicherbereich übergeben, arbeitet die Funktion wie *mem_alloc*. Ist die Größe *0*, so wird der Speicherbereich freigegeben.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Ungültige Speicheradresse. |
| `ERR_INVALID_ARGUMENT` | Ungültige Größenangabe. |
| `ERR_NOT_ENOUGH_MEMORY` | Es steht nicht genug Speicherplatz auf dem Heap oder es stehen nicht genügend physische Seitenrahmen zur Verfügung. |

### 3.7.3 *mem_free* - Größe eines Speicherbereichs ändern

**Deklaration:**

```
#include <hydrixos/mem.h>
void mem_free(void* mem);
```

**Eingabeparameter:**

| | |
|---|---|
| mem | Zeiger auf den Anfang des freizugebenden Speicherbereich. |

**Rückgabewerte:**

Keiner

**Beschreibung:**

Gibt den Speicherbereich *mem* wieder frei.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Ungültige Speicheradresse. |

## 3.8 Einfache Threadfunktionen (BaseLib-Threads)

### 3.8.1 *blthr_create* - Thread erstellen

**Deklaration:**

```
#include <hydrixos/blthr.h>
thread_t* blthr_create(void (*start)(thread_t *thr), size_t stack);

typedef struct {
    void (*func)(thread_t *thr);
    list_t ls;
}atexit_t;
typedef struct {
    sid_t      thread_sid;   /* SID des Threads */
    void*      stack;        /* Stack des Threads */
    size_t     stack_sz;     /* Größe des Thread-Stacks in Byte */
    /* Startroutine des Threads */
    void (*start)(thread_t *thr);

    atexit_t *atexit_list;     /* Liste der At-Exit-Aufrufe */
    int  atexit_cnt;           /* Anzahl der At-Exit-Aufrufe */
    mtx_t mutex;               /* Mutex des Deskriptors */
}thread_t;
```

**Eingabeparameter:**

| | |
|---|---|
| start | Zeiger auf die Initialisierugnsfunktion des Threads |
| stack | Größe des Stacks des zu erstellenden Threads (0 = Standardgröße) |

**Rückgabewerte:**

Zeiger auf den Threaddeskriptor.

NULL bei Fehlschlag.

**Beschreibung:**

Erzeugt einen neuen Thread, dessen Ausführung in der Routine *start* beginnt. Die Routine *start* erhält als Parameter einen Threaddeskriptor, der mit den Handler-Routinen der BaseLib-Threads initialisiert worden ist. Wichtig ist, dass alle unbenutzen Behandlungsroutinen mit *NULL* initialisiert werden. Der neue Thread erhält einen Stack der Größe *stack* Bytes.

Um den neuen Thread zu aktivieren, ist ein Aufruf von *blthr_awake* erforderlich.

Diese Funktion führt automatisch *blthr_cleanup* aus, um überreste von Threads, die mit *blthr_finish* beendet wurden, aufzulösen.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| ERR_INVALID_ADDRESS | Ungültige Speicheradresse. |
| ERR_NOT_ENOUGH_MEMORY | Es steht nicht genug Speicherplatz zur Erstellung des neuen Threads zur Verfügung. |

### 3.8.2 *blthr_cleanup* - Threads aufräumen

**Deklaration:**

```
#include <hydrixos/blthr.h>
void blthr_cleanup(void);
```

**Eingabeparameter:**

Keiner.

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Gibt den Stack und die Threaddeskriptoren von zuvor beendeten Threads wieder frei. Auf den Threaddeskriptor des beendeten Threads verweist der Zeiger *thr*. Hierbei werden keinerlei *atexit*-Funktionen ausgeführt. Diese Funktion sollte aufgerufen werden, wenn ein Thread sich mit *blthr_finish* beendet hat. Sie wird automatisch von *blthr_create* und *blthr_kill* ausgeführt.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Threaddeskriptor. |

### 3.8.3 *blthr_atexit* - Terminierungsfunktion für Thread hinzufügen

**Deklaration:**

```
#include <hydrixos/blthr.h>
void blthr_atexit(thread_t *thr, void (*func)(thread_t *thr));
```

**Eingabeparameter:**

| | |
|---|---|
| thr | Zeiger auf den Deskriptor des betroffenen Threads |
| func | Terminierugnsfunktion |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Fügt die Terminierungsfunktion *func* der Liste der Terminierungsfunktionen des Threads *thr* hinzu. Diese Funktion *func* wird beim Aufruf von *blthr_finish* vor Terminierung des Threads aufgerufen. Sie wird ebenfalls durch *blthr_kill* aufgerufen. Als Parameter erhält die Funktion den Parameter *thr*, welcher den Zeiger auf den zu beendenden Thread enthält.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Threaddeskriptor oder Funktionszeiger. |

### 3.8.4 *blthr_kill* - Thread beenden

**Deklaration:**

```
#include <hydrixos/blthr.h>
void blthr_kill(thread_t *thr);
```

**Eingabeparameter:**

| | |
|---|---|
| thr | Zeiger auf den Deskriptor des zu beendenden Threads. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Beendet einen Thread *thr* und gibt dessen Datenstrukturen wieder frei. Zuvor werden die mit *blthr_atexit* festgelegten Funktionen im aktuellen Threadkontext ausgeführt.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Threaddeskriptor. |
| `ERR_ACCESS_DENIED` | Die Kill-Funktion kann nicht auf den aktuellen Thread angewendet werden. |

### 3.8.5 *blthr_finish* - Aktuellen Thread terminieren

**Deklaration:**

```
#include <hydrixos/blthr.h>
void blthr_finish(void);
```

**Eingabeparameter:**

Keine.

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Terminiert den aktuellen Thread, indem der Thread entgültig angehalten wird. Ein anderer Thread des Prozesses muss jedoch dessen Datenstrukturen durch den Aufruf von *blthr_cleanup* nachträglich freigeben.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Threaddeskriptor. |

### 3.8.6 *blthr_init* - Initialisierungsroutine des Threads

**Deklaration:**

```
#include <hydrixos/blthr.h>
void blthr_init(thread_t *thr);
```

**Eingabeparameter:**

| | |
|---|---|
| thr | Zeiger auf die eigene Thread-Datenstruktur |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Diese Routine wird eigentlich durch das Threadpaket nach dem architekturspezifischen Start-up-Code des neuen Threads aufgerufen. Sie initialisiert den neuen Thread *thr* vollständig und ruft die bei *blthr_create* definierte Startroutine auf. Sie ist üblicherweise eine interne Routine des Threadpakets, kann aber für die Entwicklung eines eigenen Threadpakets nützlich sein.

Sie ist jedoch nicht zwingend automatisch die Routine, die der Kernel als Start-up-Code für den Thread verwendet. Hierfür kann durch die Bibliothek intern eine architekturspezifische Routine mit Namen *blthr_init_arch* verwendet werden.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_ACCESS_DENIED` | Thread bereits initialisiert. |

### 3.8.7 *blthr_yield* - CPU-Zeit abgeben

**Deklaration:**

```
#include <hydrixos/blthr.h>
void blthr_yield(sid_t sid);
```

**Eingabeparameter:**

| | |
|---|---|
| sid | SID eines Threads, der die restliche CPU-Zeit erhalten soll. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Der Aufruf dieser Routine führt dazu, dass der aktuelle Thread augenblicklich seine aktuelle CPU-Zeit verliert. Benennt er im Parameter *sid* einen weiteren Thread, so erhält dieser seine verbleibende CPU-Zeit. (*Näheres siehe hymk_yield_thread*)

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültige SID. |

### 3.8.8 *blthr_awake* - Thread aufwecken

**Deklaration:**

```
#include <hydrixos/blthr.h>
void blthr_awake(thread_t *thr);
```

**Eingabeparameter:**

| | |
|---|---|
| thr | Zeiger auf den Deskriptor des Threads, der aufgeweckt werden soll. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Diese Routine weckt einen Thread auf, auf den der Deskriptor *thr* zeigt, sofern er zuvor mit *blthr_freeze* eingefroren wurde.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Threaddeskriptor. |

### 3.8.9 *blthr_freeze* - Thread anhalten

**Deklaration:**

```
#include <hydrixos/blthr.h>
void blthr_freeze(thread_t *thr);
```

**Eingabeparameter:**

| | |
|---|---|
| thr | Zeiger auf den Deskriptor des Threads, der angehalten werden soll. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Diese Routine hält den durch den Deskriptor *thr* bezeichneten Thread an.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Threaddeskriptor. |

## 3.9 Synchronisation mit Mutexen

### 3.9.1 *MTX_DEFINE* - Globalen Mutex definieren

**Deklaration:**

```
#include <hydrixos/mutex.h>
#define MTX_DEFINE
```

**Eingabeparameter:**

Keiner.

**Rückgabewerte:**

Initialisierer für Mutexe, die auf globaler Ebene defniert werden.

**Beschreibung:**

Dieser Aufruf liefert einen Initialisierungswert für einen neuen Mutex zurück.

**Beschränkungen:**

Keine.

**Fehlercodes:**

Keine.

### 3.9.2 *MTX_NEW* - Neuen Mutex initialisieren

**Deklaration:**

```
#include <hydrixos/mutex.h>
#define MTX_NEW()
```

**Eingabeparameter:**

Keiner.

**Rückgabewerte:**

Initialisierer für einen neuen Mutex, der auf lokaler Ebene definiert oder initialisiert werden soll.

**Beschreibung:**

Dieses Makro liefert einen Initialisierer für einen neuen Mutex zurück.

**Beschränkungen:**

Keine.

**Fehlercodes:**

Keine.

### 3.9.3 *mtx_trylock* - Mutex schließen, wenn möglich

**Deklaration:**

```
#include <hydrixos/mutex.h>
int mtx_trylock(mtx_t *mutex);
```

**Eingabeparameter:**

| | |
|---|---|
| mutex | Mutex, der geschlossen werden soll. |

**Rückgabewerte:**

| | |
|---|---|
| 1 | falls der Mutex geschlossen wurde |
| 0 | falls der Mutex nicht geschlossen werden konnte |

**Beschreibung:**

Dieser Aufruf versucht den Mutex *mutex* zu schließen, ohne jedoch in eine Warteschleife einzutreten.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Mutex. |

### 3.9.4 *mtx_lock* - Mutex schließen oder warten

**Deklaration:**

```
#include <hydrixos/mutex.h>
int mtx_lock(mtx_t *mutex, long timeout);
```

**Eingabeparameter:**

| | | | |
|---|---|---|---|
| mutex | Mutex, der geschlossen werden soll. | | |
| timeout | Anzahl der Versuche den Mutex zu schließen | | |
| | MTX_NO_TIMEOUT | 0 | Nicht auf Freigabe des Mutexes warten |
| | MTX_UNLIMITED | -1 | Unendlich auf Mutex-Freigabe warten. |

Anmerkung: Die Werte 0 und -1 dürfen auch direkt verwendet werden.

**Rückgabewerte:**

| | |
|---|---|
| 1 | falls der Mutex geschlossen wurde |
| 0 | falls der Mutex nicht innerhalb des Timeouts geschlossen werden konnte |

**Beschreibung:**

Dieser Aufruf versucht den Mutex *mutex* zu schließen. Ist es nicht auf Anhieb möglich, so gibt der Thread mittels *blthr_yield* seine CPU-Zeit ab und nimmt anschließend *timeout* erneute Versuche , den Mutex zu schließen.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| ERR_INVALID_ARGUMENT | Ungültiger Mutex. |
| ERR_TIMED_OUT | Timeout des Mutex wurde erreicht. |

### 3.9.5 *mtx_unlock* - Mutex freigeben

**Deklaration:**

```
#include <hydrixos/mutex.h>
void mtx_unlock(mtx_t *mutex);
```

**Eingabeparameter:**

| | |
|---|---|
| mutex | Mutex, der freigegeben werden soll. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Dieser Aufruf gibt den Mutex *mutex* wieder frei. Der Aufruf sollte nur auf einen Mutex angewendet werden, der zuvor durch den selben Thread geschlossen wurde!

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Mutex. |

## 3.10 Datentypen der hyBaseLib

Die hyBaseLib baut auf folgenden Datentypen auf (deklariert in „*hydrixos/types.h*“):

| | |
|---|---|
| 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) |
| mtx_t | Mutex-Datentyp |
| thread_t | Thread-Deskriptor |
| atexit_t | Liste von At-Exit-Aufrufen |
| region_t | Region-Deskriptor |
| block_t | Block auf dem Heap (interne Datenstruktur) |
| tls_t | TLS-Variable (identisch mit uint32_t) |
| utf8_t | Byte aus einem UTF-8-String (identisch mit *int8_t*) |

Auf Systemen mit hardware- oder compilerseitigen 64-bit Unterstützung stehen die Datentypen *uint64_t* und *int64_t* zusätzlich zur Verfügung.

## 3.11 Globale Variablen der hyBaseLib

errno_t* tls_errno;  Zegier auf die Errno-Variable auf dem TLS. (hydrixos/error.h)

region_t *regions;  Verkettete Liste der Speicherregionen (hydrixos/mem.h).

region_t *code_region;  Deskriptor der Code-Region
region_t *data_region;  Deskriptor der Daten-Region
region_t *stack_region;  Deskriptor der Stack-Region
region_t *heap_region;  Deskriptor der Heap-Region

mtx_t reg_mutex;  Mutex, der die Regionstabelle schützt (hydrixos/mem.h).

thread_t **tls_my_thread;  Zeiger auf Datenstruktur des aktuellen Threads (hydrixos/blthr.h)

## 3.12 Verwaltung der Mapping-Region

### 3.12.1 *pmap_alloc* - Speicher für Page-Mapping alloziieren

**Deklaration:**

```
#include <hydrixos/pmap.h>
void* pmap_alloc(size_t sz);
```

**Eingabeparameter:**

| | |
|---|---|
| sz | Größe des zu alloziierenden Speicherbereichs in Bytes. |

**Rückgabewerte:**

Zeiger auf den neu alloziierten Speicherbereich. *NULL* bei Fehlschlag.

**Beschreibung:**

Dieser Aufruf alloziiert einen Speicherbereich bestehend aus *sz* Bytes in der Mapping-Region. Dabei wird der alloziierte Speicherbereich jedoch nicht mit physischen Seitenrahmen unterlegt. Dieser Speicherbereich kann für Page-Mappings verwendet werden.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Größenangabe. |
| `ERR_NOT_ENOUGH_MEMORY` | Zu wenig Heap für die Mapping-Verwaltung oder zu wenig virtueller Speicher in der Mapping-Region. |

### 3.12.2 *pmap_free* - Speicher von Page-Mapping freigeben

**Deklaration:**

```
#include <hydrixos/pmap.h>
void pmap_free(void* ptr);
```

**Eingabeparameter:**

| | |
|---|---|
| ptr | Zeiger auf den freizugebenden Speicherbereich. |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Dieser Aufruf gibt einen Speicherbereich auf den der Zeiger *ptr* in der Mapping-Region wieder frei. Die Speicherseiten werden aus dem Adressbereich dabei entfernt.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Zeiger. |

### 3.12.3 *pmap_mapalloc* - Mapping-Speicher alloziieren und mit Speicherseiten unterlegen

**Deklaration:**

```
#include <hydrixos/pmap.h>
void* pmap_mapalloc(size_t sz);
```

**Eingabeparameter:**

| | |
|---|---|
| sz | Größe des zu alloziierenden Speicherbereichs in Byte. |

**Rückgabewerte:**

Zeiger auf den neu alloziierten Speicherbereich. *NULL* bei Fehlschlag.

**Beschreibung:**

Dieser Aufruf alloziiert einen Speicherbereich bestehend aus *sz* Bytes in der Mapping-Region. Dabei wird der alloziierte Speicherbereich auch mit physischen Seitenrahmen unterlegt.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Größenangabe. |
| `ERR_NOT_ENOUGH_MEMORY` | Zu wenig Heap für die Mapping-Verwaltung, zu wenig virtueller Speicher in der Mapping-Region oder kein physischer Speicher mehr zur Verfügung. |

## 3.13 Der XML-Parser

### 3.13.1 *spxml_replace_stdentities* - XML-Standardentities durch UTF-8-Zeichen ersetzen

**Deklaration:**

```
#include <hydrixos/spxml.h>
utf8_t* spxml_replace_stdentites(const utf8_t *text, size_t len);
```

**Eingabeparameter:**

| | |
|---|---|
| text | Zu durchsuchender Text |
| len | Länge des Text in Byte |

**Rückgabewerte:**

Zeiger auf die veränderte Kopie (NULL bei Fehlschlag)

**Beschreibung:**

Dieser Aufruf ersetzt alle XML-Standardentities in einem Text *text* der Länge *len* durch ihre UTF-8-Entsprechungen und speichert diese in einem getrennten Speicherbereich. Ersetzt werden < durch <, > durch >; & durch &, " durch “ und ' durch ’.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Text-Zeiger |

### 3.13.2 *spxml_create_tree* - XML-Daten in Baum umwandeln

**Deklaration:**

```
#include <hydrixos/spxml.h>
typedef struct spxml_node_st
{
        int             type;           /* SPXML Elementtyp */

        const utf8_t*        tag;            /* Zeiger zum XML-Tag (ohne < > o.ä.) */
        size_t                tag_len;        /* Länge des Tags */

        struct spxml_node_st    *children;      /* Liste der Kind-Elemente */
        list_t                  ls;             /* Liste der Schwesternelemente */
}spxml_node_t;
        /* <!-- Kommentar --> */
#define SPXMLEVENT_COMMENT              0
        /* <? Processing Informations ?> */
#define SPXMLEVENT_PROCESSING           1
        /* <Leerer_Tag/> */
#define SPXMLEVENT_EMPTY_TAG            2
        /* </End_Tag> */
#define SPXMLEVENT_END_TAG              3
        /* <Begin_Tag> */
#define SPXMLEVENT_BEGIN_TAG            4
        /* Daten */
#define SPXMLEVENT_DATA                 5
        /* Element enthält nur Whitespaces (LF, CR, TAB, SPACE) */
#define SPXMLEVENT_WHITESPACE           6
        /* Ende der Datei */
#define SPXMLEVENT_EOF                  7
        /* Ungültiges Element */
#define SPXMLEVENT_INVALID              8
const utf8_t* spxml_create_tree(const utf8_t *xml, size_t len, spxml_node_t *node);
```

**Eingabeparameter:**

| | |
|---|---|
| xml | Zu durchsuchender Text |
| len | Länge des Text in Byte |
| node | Anfangsknoten |

**Rückgabewerte:**

Zeiger auf das Ende des analysierten Textes, NULL bei Fehlschlag

**Beschreibung:**

Dieser Aufruf durchsucht den Text *xml* (der Länge *len* Byte) nach einer XML-Anweisung. Wird eine Anweisung gefunden, so wird der Knoten *node* entsprechend der Inhalte des Elements konfiguriert. Findet sich ein untergeordneter Knoten, so wird dieser rekursiv mittels *spxml_create_tree* ebenfalls durchsucht und der *children*-Liste von *node* hinzugefügt. Zurückgegeben wird die Textposition, an der das Parsen abgebrochen wurde, weil das Tag entweder kein gegenseitig abschließendes Tag (also zu <Text> z.B. </Text) hat, oder das Tag nur ein Einzelelement ohne Unterelement ist.
Wichtig ist, dass der Zeiger *tag* des Elements *node* (sowie die entsprechenden Zeiger aller Kindelemente) auf Positionen im String *xml* zeigen. Folglich darf der String erst dann an eine andere Adresse verschoben oder freigegeben werden, wenn der XML-Baum beseitigt wurde.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| ERR_INVALID_ARGUMENT | Der XML-Code ist nicht einwandfrei. |
| ERR_INVALID_ADDRESS | Einer der Eingabezeiger ist ungültig. |

### 3.13.3 *spxml_destroy_tree* - XML-Baum vernichten

**Deklaration:**

```
#include <hydrixos/spxml.h>
void spxml_destroy_tree(spxml_node_t *node)
```

**Eingabeparameter:**

| | |
|---|---|
| node | Anfangsknoten des zu löschenden Baums |

**Rückgabewerte:**

Keiner.

**Beschreibung:**

Dieser Aufruf löscht alle Kind-Knoten von *node* aus dem Speicher und entfernt die entsprechenden *children*-Eintragung in *node*. Die Struktur *node* selbst wird dabei jedoch nicht freigegeben.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ARGUMENT` | Ungültiger Zeiger. |

### 3.13.4 *spxml_resolve_path* - XML-Pfad in Element auflösen

**Deklaration:**

```
#include <hydrixos/spxml.h>
spxml_node_t* spxml_resolve_path(const utf8_t *path, spxml_node_t *node)
```

**Eingabeparameter:**

| | |
|---|---|
| path | Aufzulösender Pfad |
| node | Anfangsknoten des zu durchsuchenden Baums |

**Rückgabewerte:**

Gefundenes Element. NULL, bei Fehlschlag.

**Beschreibung:**

Dieser Aufruf verwendet den Pfad *path* um ein Element im XML-Baum *node* zu finden. Der Pfad ist ähnlich einem Dateipfad aufgebaut (Wurzel /, Pfad-Trennzeichen /), wobei jedes Element des Pfads den Namen des entsprechenden XML-Elements trägt.

**Beschränkungen:**

Keine.

**Fehlercodes:**

| | |
|---|---|
| `ERR_INVALID_ADDRESS` | Ungültiger Zeiger. |
| `ERR_INVALID_ARGUMENT` | Pfad nicht gefunden oder ungültig. |

# Teil III

# Implementierung der hyBaseLib

Fehlt.