Struktur des Lehr-Betriebssystems Topsy und seiner Entwicklungsumgebung

Studienarbeit im Fach Informatik

vorgelegt von

Marco Pfattner

geb. am 2. Oktober 1977 in Bamberg

Angefertigt am

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

Betreuer:	***Prof. Dr. Fridolin Hofmann*** ***Dr. Frank Bellosa***
Beginn der Arbeit:	15. März 2000
Abgabe der Arbeit:	14. Juni 2000

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 14. Juni 2000, _____

Inhaltsverzeichnis

Kapitel 1

Einleitung

Topsy ist ein 32 Bit Betriebssystem, das zu Lehrzwecken an der ETH Zürich entwickelt wurde. Einige Vorteile dieses Betriebssystems sind:

- Einfacher Aufbau: Der Betriebssystem-Kern ist klein und enthält nur die wesentlichen Funktionen.
- Weitgehende Hardwareunabhängigkeit: Nur wenige Teile des Betriebssystems sind hardwareabhängig und in Assembler programmiert. Die hardwareabhängigen Teile sind klar von den hardwareunabhängigen Teilen getrennt.
- Verständlicher und lesbarer Programmcode: Topsy wurde größtenteils in C programmiert. Die enthaltenen Algorithmen wurden nicht auf Geschwindigkeit hin optimiert, sondern möglichst einfach gehalten.

Trotz des einfachen, modularen Aufbaus enthält Topsy alle wesentlichen Elemente, die sich auch in einem "großen" Betriebssystem finden, wie zum Beispiel Speichermanagement, Threadmanagement, Interprozesskommunikation oder Gerätetreiber.

Ein weiterer Vorteil von Topsy ist die Verfügbarkeit des Quellcodes. Natürlich sind auch die Quellen anderer, leistungsfähigerer Betriebssysteme erhältlich (z. B. Linux), aber diese sind teilweise unübersichtlich und durch ihre Größe und Komplexität nicht zu Lehrzwecken geeignet.

Neben Topsy existieren noch weitere Lehrbetriebssysteme, wie z. B. MINIX [Tan90b], Nachos [Nac] oder XINU [Com88].

Topsy kann zwei Adressräume verwalten. Es gibt einen Kernadressraum und einen Benutzeradressraum. In den verschiedenen Adressräumen können dann Threads ausgeführt werden. Eine Besonderheit von Topsy ist der aus Threads

aufgebaute Kern. Damit ist es beispielsweise möglich, dass mehrere Benutzerprogramme gleichzeitig verschiedene System-Calls aufrufen. Während ein Benutzer-Thread Speicher allokieren will und somit den Speichermanager beschäftigt kann ein anderer Thread mithilfe des Threadmanagers einen neuen Thread starten. Ein weiterer Vorteil ist, dass sich die verschiedenen Teile des Kerns mit einem Messagepassing-Mechanismus verständigen können.

Topsy arbeitet mit 32 Bit langen, virtuellen Adressen. Damit können also 4 GB adressiert werden. Der Benutzeradressraum ist dabei auf die unteren 2 GB beschränkt, der Kernadressraum erhält 512 MB. Die restlichen 1536 MB werden nicht verwendet. Mithilfe der MMU können die virtuellen Adressen auf physikalische Adressen abgebildet werden.

Da Topsy kein Dateisystem besitzt, müssen der Betriebssystem-Kern und die Benutzerprogramme statisch gebunden werden. Ein Nachladen von Modulen ist noch nicht möglich.

1.1 Struktur von Topsy

Das Betriebssystem Topsy ist modular aufgebaut. Abbildung 1.1 gibt einen Überblick über die Struktur von Topsy.

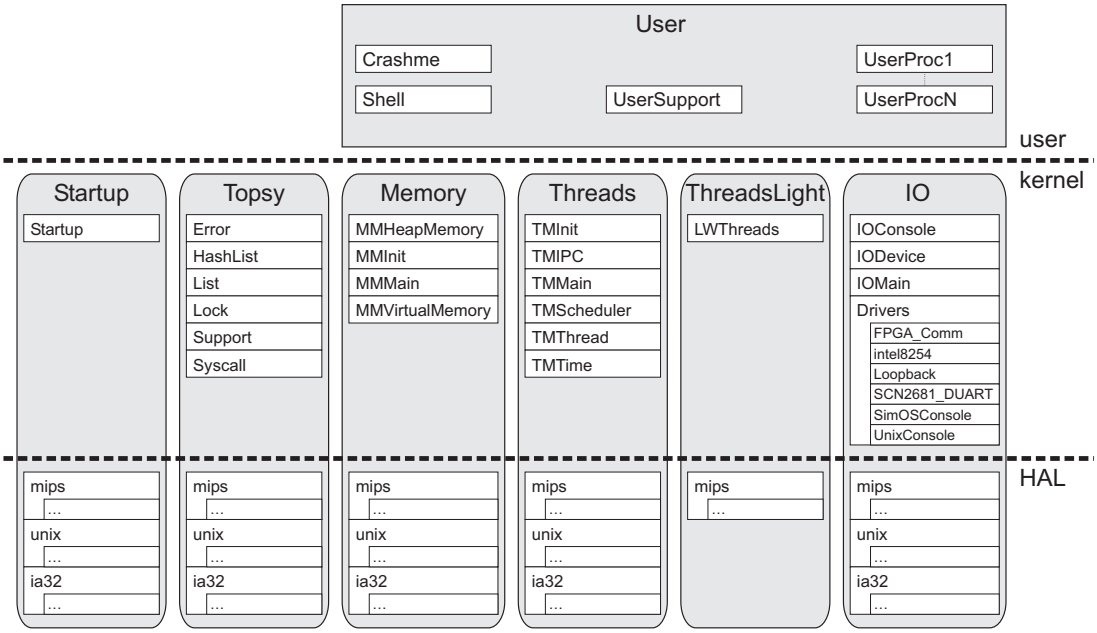

Abbildung 1.1: Modulare Struktur von Topsy

Der Kern besteht aus drei Hauptmodulen: Memory, Threads und IO. Diese Module sind unabhängig voneinander als Threads implementiert, die im Ker-

nadressraum ablaufen. Die Kommunikation zwischen Threads wird durch einen Messagepassing-Mechanismus ermöglicht.

Das Modul ThreadsLight bietet in etwa dieselbe Funktionalität wie das Modul Threads. Es gibt jedoch einige Einschränkungen beim Nachrichtenaustausch und beim Scheduling.

Das Modul Startup kümmert sich um den Bootvorgang von Topsy. Hier werden verschiedene Initialisierungen vorgenommen und verschiedene Kern-Threads gestartet.

Das Modul Topsy stellt verschiedene Routinen zur Verfügung, die von anderen Teilen des Kerns benutzt werden. Dieses Modul stellt auch die Verbindung zwischen Benutzerprogrammen und dem Kern dar. Durch System-Calls haben Benutzerprogramme die Möglichkeit, auf Funktionen des Kerns zurückzugreifen um beispielsweise einen neuen Thread zu erzeugen oder Nachrichten zu empfangen.

Jedes dieser Kern-Module besteht aus einem hardwareunabhängigen sowie einem hardwareabhängigen, teilweise in Assembler programmierten Teil. Diese beiden Teile sind klar voneinander getrennt. Der Hardware Abstraction Layer (HAL) bildet eine Schnittstelle zwischen der Hardware-Architektur (mips, ia32, ...) und dem eigentlichen Kern.

Auf der obersten Ebene laufen die Benutzerprogramme. Der Hauptthread auf dieser Ebene ist normalerweise die Shell, in welche der Benutzer Befehle eingeben und somit neue Threads starten kann.

In Topsy laufen nur zwei Prozesse: Ein Kernprozess mit seinen verschiedenen Threads und ein Benutzerprozess. Alle anderen Benutzerprogramme werden als Threads gestartet.

1.2 Überblick

Abbildung 1.2 gibt einen detaillierten Überblick über die Struktur und die Zusammenhänge der einzelnen Module von Topsy.

Auf der linken Seite befindet sich eine Liste von System-Calls, mit deren Hilfe Benutzerprogramme und auch Teile des Kerns auf Funktionen des Kerns zugreifen können.

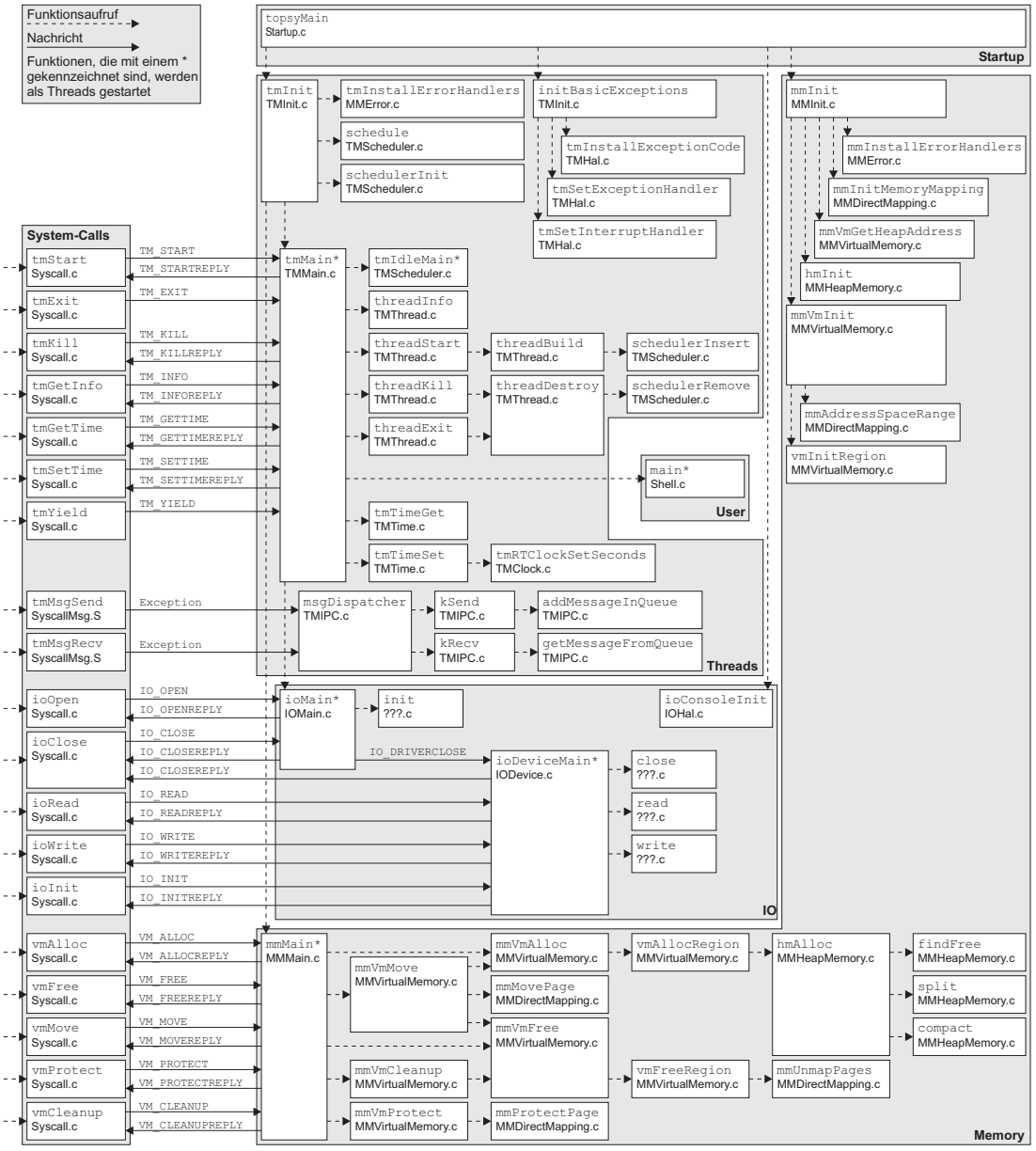

Abbildung 1.2: Überblick über Topsy

Kapitel 2

Laufumgebung

Topsy läuft derzeit auf einem MIPS R3000 Prozessor. Bemühungen, Topsy an die i386-Architektur anzupassen, sind im Gange und sollten in Kürze abgeschlossen sein [Ruf98].

2.1 MIPS R3000

Der MIPS R3000 ist ein 32 Bit RISC Prozessor. Der CPU Kern besitzt eine fünfstufige Pipeline (Abbildung 2.1) und 32 Register mit jeweils 32 Bit. Der MIPS R3000 besitzt auch noch einen Koprozessor, der sich um die Exceptions kümmert und einen voll assoziativen TLB (Translation Lookaside Buffer) mit 64 Einträgen besitzt (siehe Abschnitt 7.3). Der TLB kann Zuordnungen von virtuellen zu physikalischen Seiten speichern. Der Prozessor kann maximal 4 GB physikalischen Speicher verwalten.

Der Prozessor besitzt einen 4 kB (je nach Modell auch 8 kB) großen Instruktionscache, welcher mit physikalischen Adressen arbeitet und einen 2 kB großen Datencache, welcher einen write through Mechanismus verwendet, um den physikalischen Speicher aktuell zu halten, d. h. jede Änderung im Cache hat zur Folge, dass auch der langsamere Arbeitsspeicher aktualisiert wird. Damit das System dadurch nicht ausgebremst wird, werden einige Schreib-Operationen gesammelt und zu einem späteren Zeitpunkt ausgeführt.

Dieser relativ billige Prozessor mit geringem Stromverbrauch eignet sich zum Einsatz in eingebetteten Systemen. Ein Einsatzgebiet dieses Prozessors sind beispielsweise Laser-Drucker.

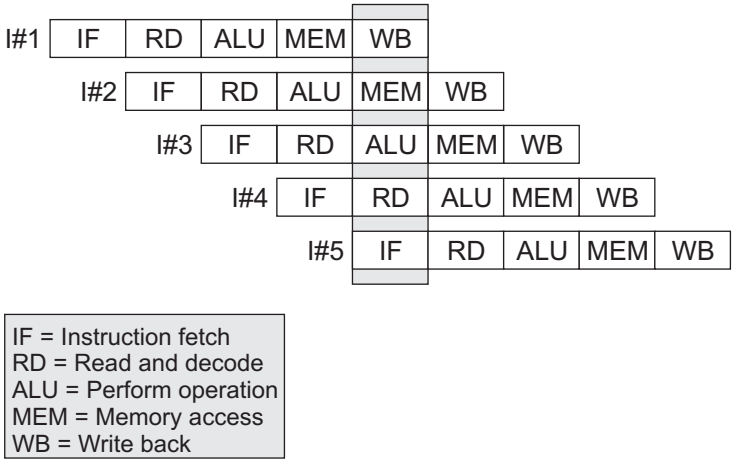

Abbildung 2.1: Fünfstufige Pipeline des MIPS R3000

2.2 Simulationsumgebung

Topsy kann auch auf dem MIPS R3000 Simulator MipsSim ausgeführt werden. Der Simulator wurde komplett in Java geschrieben und läuft somit auf den meisten Systemen.

Der Simulator stellt einen MIPS R3000 Prozessor mit 8 MB Hauptspeicher und zwei seriellen Schnittstellen zur Verfügung. Eine der seriellen Schnittstellen wird verwendet, um Text auf der Konsole auszugeben und um die Benutzereingaben einzulesen.

Der Simulator erwartet eine Datei mit einem lauffähigen Betriebssystem. Diese Datei muss im Motorola S-record Format (siehe Abschnitt 3) vorliegen.

MipsSim kann mit verschiedenen Optionen gestartet werden:

-t tracefile: Die Operationen, die vom MIPS-Simulator durchgeführt werden, werden in der Datei tracefile gespeichert. Die Ausgabe erfolgt dabei in Assembler-Code.

-s symbolfile: Kann nur zusammen mit der Option -t verwendet werden. Dabei werden die Symbole eingelesen und in tracefile ausgegeben.

-g: Schnittstelle zum GNU Debugger wird aktiviert.

-v: Verbose. Auf der Konsole werden verschiedene Meldungen ausgegeben, beispielsweise wenn ein Interrupt ausgelöst wird oder wenn ein System-Call aufgerufen wird.

-b: Big Endian

-l: Little Endian

2.2.1 Schnittstelle zum GNU Debugger

Durch die Aktivierung der Schnittstelle zum GNU Debugger wird das Debugging von Topsy ermöglicht. Allerdings kann nicht der normale GNU Debugger verwendet werden, sondern eine spezielle Version (gdb-simos), die eigentlich für den MIPS Simulator SimOS vorgesehen ist [Sima]. Da jedoch die Schnittstelle zum GNU Debugger bei den Simulatoren SimOS und MipsSim gleich ist, kann auch für den MipsSim dieser Debugger verwendet werden. Eine vorcompilierte Version von gdb-simos für Sparc-Architekturen kann von [Simb] kopiert werden.

Nach dem Starten von Topsy mit der Option -g und dem Starten des SimOS GNU Debuggers vom Topsy-Verzeichnis aus müssen zunächst die Symbole in den Debugger eingelesen werden und es muss eine Verbindung zu Topsy hergestellt werden. Dies geschieht durch folgende Befehle:

```
(gdb) file topsy.ecoff
(gdb) target simos localhost:2345
```

Jetzt kann mit dem Debugger normal weitergearbeitet werden. Beispielsweise können Dateien oder Funktionen mit dem Befehl `list` angezeigt werden:

```
(gdb) list mmUTLBError
124     static void mmUTLBError(ThreadId currentThread)
125     {
126       Register lo=0, hi=0;
127
128       unsigned int virt_page=0;
129       unsigned int phys_page=0;
130
131       Frame_Desc *frame;
132       unsigned int page;
```

Mit dem Befehl `break` können Breakpoints in der aktuellen Datei gesetzt werden:

```
(gdb) break 142
Breakpoint 1 at 0x800235cc: file Memory/mips/MMError.c, line 142.
```

Weitere Informationen zum Umgang mit dem SimOS GNU Debugger gibt es unter [GDB].

Kapitel 3

Das Modul Startup

Beim Starten des Rechners wird der Programmzähler mit einer bestimmten Adresse initialisiert. Diese Adresse zeigt normalerweise auf eine Stelle im nichtflüchtigen Speicher (ROM, EPROM, ...). An dieser Speicherstelle befindet sich ein Code, der das Betriebssystem von einem externen Speicher (Festplatte, Netzwerk, ...) lädt.

Der Lader erwartet, dass er das Betriebssystem in einem bestimmten Format vorfindet. Der Lader des MIPS-Prozessors verwendet dazu das Motorola S-record Format. Der Simulator benötigt eine gültige S-record Datei (topsy.srec). Jede Zeile dieser Datei wird als Record interpretiert.

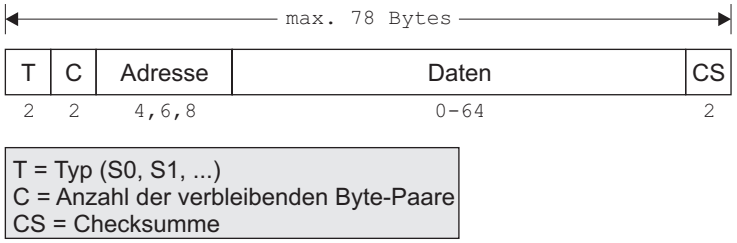

Abbildung 3.1: S-record

Ein Record besteht aus einem Typ (S0, S1, S2, S3, S5, S7, S8 oder S9), einer Größenangabe welche die Anzahl der verbleibenden Byte-Paare angibt, einer Adresse an welche die Daten geladen werden sollen, den Daten und einer Prüfsumme. Für 32 Bit Systeme werden normalerweise Records vom Typ S3 und S7 verwendet.

Bei S3 Records wird die Adresse als 4 Byte Adresse (also 32 Bits) interpretiert. Das Datenfeld enthält die Daten, welche an die angegebene Adresse geladen werden sollen. Ein Record vom Typ S7 enthält eine 4 Byte lange Startadresse, an welche der Lader nach dem Laden springen soll. Ein S7 Record enthält keine Daten.

Genauere Informationen zum S-record Format gibt es unter [SREa] und [SREb].

Die Datei topsy.srec enthält also viele S3 Records und am Ende ein S7 Record. Die Adresse im S7 Record zeigt auf eine kleine Assembler-Routine (`__start` in Startup/mips/start.S). Die Funktion `__start` bereitet einen Bootstack vor, der später als Exception-Stack verwendet wird. Anschließend wird die Funktion `StartMeFirst` aufgerufen, welche dann die Funktion `topsyMain` aufruft.

Abbildung 3.2 zeigt, was beim Ausführen der Funktion `topsyMain` geschieht.

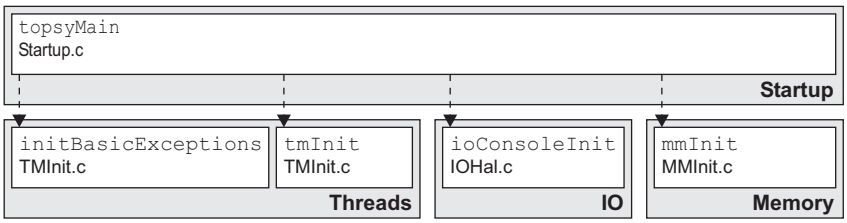

Abbildung 3.2: Startup

In dieser Funktion werden der Exception-Handler, die Konsole, der Speichermanager und der Threadmanager initialisiert. Die Funktion `tmInit` überträgt die Kontrolle dem ersten Kern-Thread und startet noch einen initialen Benutzer-Thread, z. B. die Shell.

Der Speichermanager (`mmMain`), Threadmanager (`tmMain`), IO-Manager (`ioMain`) sowie sämtliche Treiber werden beim Booten als Threads gestartet und laufen anschließend in eine Endlosschleife, um Befehle entgegenzunehmen und zu bearbeiten.

Was beim Aufrufen der Initialisierungsfunktionen `tmInit` und `mmInit` geschieht, wird in den Kapiteln Threads und Memory beschrieben.

Kapitel 4

Das Modul Threads

Threads werden nicht wie Prozesse in einem eigenen Adressraum ausgeführt, sondern sie laufen im Adressraum des aufrufenden Threads. In Topsy gibt es nur zwei Adressräume – den Kernadressraum und den Benutzeradressraum.

Anders als bei vielen anderen Betriebssystemen ist auch der Kern aus einzelnen Threads aufgebaut. Das bietet den Vorteil, dass mehrere System-Calls gleichzeitig bearbeitet werden können und dass Kern-Threads genauso wie Benutzer-Threads Nachrichten austauschen können. Auch die System-Calls werden durch Nachrichten an die verschiedenen Kern-Threads implementiert (siehe Abschnitt 8.1).

Das Modul Threads bietet die Möglichkeit, Threads zu starten und zu beenden sowie Mechanismen zum Austausch von Nachrichten. In diesem Modul befindet sich auch der Scheduler, der bestimmt, wann welcher Thread ausgeführt wird und in welcher Reihenfolge (Priorität) diese Threads ausgeführt werden.

Abbildung 4.1 zeigt, in welchen Zuständen sich ein Thread befinden kann und welche Zustandsübergänge möglich sind.

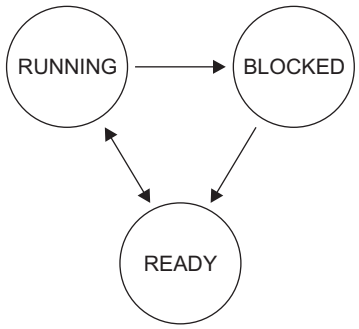

Abbildung 4.1: Zustände von Threads

Der Zustand RUNNING bedeutet, dass der Thread momentan ausgeführt wird. Der Thread wechselt in den Zustand BLOCKED, wenn er beispielsweise auf den Empfang einer Nachricht wartet oder er wechselt in den Zustand READY, wenn er von einem anderen, höherprioren Thread verdrängt wird. Der Thread wechselt vom Zustand READY in den Zustand RUNNING, wenn er vom Scheduler wieder dem Prozessor zugeordnet wird. Wenn ein Thread durch das Verlassen seiner Hauptfunktion oder durch Aufruf der Funktion `tmExit` beendet wird, verlässt er den Zustand RUNNING. Ein Thread verlässt die Zustände READY und BLOCKED, wenn er von einem anderen Thread terminiert wird.

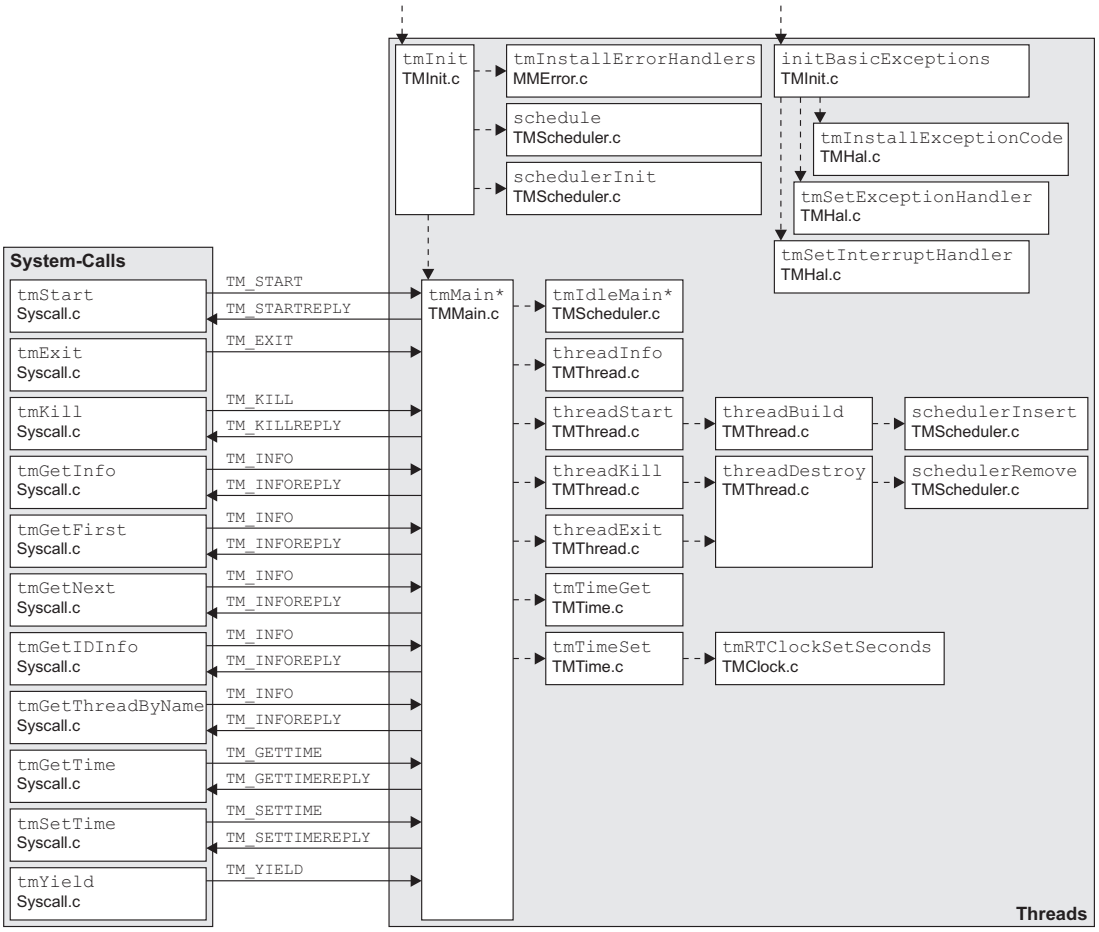

Abbildung 4.2: Überblick über das Modul Threads

Abbildung 4.2 gibt einen Überblick über das Modul Threads und die bereitgestellten System-Calls.

4.1 Initialisierung

Mit einem Aufruf von `tmInit` werden verschiedene Strukturen zur Verwaltung der Threads initialisiert. Als erstes wird der Scheduler initialisiert. Diesem wird als Argument der erste Thread (`tmThread`) übergeben. Anschließend werden neue Threads erzeugt: der Speichermanager und der Threadmanager.

Der Threadmanager (`tmMain`) startet wiederum einige Threads: einen für das IO-Processing, einen Benutzerprozess und einen Idle-Thread. Die Funktion `tmMain` läuft anschließend in eine Endlosschleife, um System-Calls zu bearbeiten. Die verschiedenen System-Calls zum Erzeugen und Beenden von Threads, Abrufen von Informationen über Threads und Abfragen der Zeit werden hier empfangen, bearbeitet und die Antwort wird dann zurückgeschickt.

Um schnell auf die Informationen der einzelnen Threads (z. B. Name, ID) zugreifen zu können, werden die Zeiger auf diese Informationen in einer Hash-Tabelle verwaltet. Auf diese Zeiger kann dann mittels Thread-ID zugegriffen werden. Die Zeiger auf die Thread-Strukturen werden zusätzlich in einer verketteten Liste verwaltet.

Nach der Initialisierung dieser Strukturen und Erzeugung der Threads wird dem Scheduler die Aufgabe überlassen, welcher Thread nun als nächstes dem Prozessor zugeordnet wird.

4.2 Kommunikation zwischen Threads

Damit die einzelnen Threads miteinander kommunizieren können (IPC, Inter Process Communication), gibt es einen Messagepassing-Mechanismus. Ein Thread hat somit die Möglichkeit, Nachrichten an andere Threads zu senden bzw. Nachrichten zu empfangen. Ein Message-Dispatcher im Kern sorgt dafür, dass die einzelnen Nachrichten an den richtigen Empfänger weitergeleitet werden.

Jeder Thread besitzt eine eigene Message-Queue, in welcher ankommende Nachrichten gespeichert werden. Diese Message-Queue hat eine feste Anzahl von Plätzen und kann deshalb nicht beliebig viele Nachrichten aufnehmen.

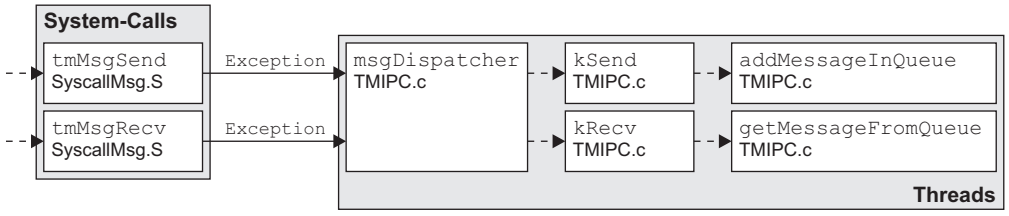

Abbildung 4.3: System-Calls zum Senden und Empfangen von Nachrichten

Wie Abbildung 4.3 zeigt, können zum Senden und Empfangen von Nachrichten die beiden System-Calls `tmMsgSend` und `tmMsgRecv` verwendet werden. Diese Funktionen werden in der Datei Topsy/Syscall.h zur Verfügung gestellt. Bei einem Aufruf einer dieser Funktionen wird eine Exception ausgelöst, wodurch die Funktion `msgDispatcher` aufgerufen wird. Der Message-Dispatcher versucht dann, beim Senden der Nachricht diese in die Message-Queue einzufügen bzw. beim Empfangen aus dieser Queue eine Nachricht zu lesen.

4.2.1 Senden von Nachrichten

Die Funktion `tmMsgSend` ist wie folgt definiert:

```
SyscallError tmMsgSend (ThreadId to, Message *msg);
```

`to`: ID des Threads, an welchen die Nachricht geschickt werden soll.

`msg`: Zeiger auf eine Struktur vom Typ `Message`. Diese Struktur enthält die Thread-ID des Absenders, eine Nachrichten-ID, welche den Typ der Nachricht bestimmt und schließlich noch den eigentlichen Inhalt der Nachricht (`msg`).

Die verschiedenen Nachrichten-Typen und Strukturen sind in der Datei Topsy/Messages.h definiert.

```
typedef enum {
  ANYMSGTYPE, TM_START, TM_EXIT, ...
} MessageId;

typedef struct UserMessage_t {
  void* p1;
  void* p2;
  void* p3;
} UserMessage;

typedef union SpecMsg_u {
  UserMessage userMsg;
  TMStartMsg tmStart; TMStartReplyMsg tmStartReply;
  TMKillMsg tmKill; TMKillReplyMsg tmKillReply;
  ...
  SyscallReply syscallReply;
} SpecMessage;
```

```
typedef struct Message_t {
  ThreadId from;
  MessageId id;
  SpecMessage msg;
} Message;
```

Die Variable `msg` der Struktur `Message` ist vom Typ `SpecMessage`. Dieser Typ ist ein Verbund aller verschiedenen Nachrichten-Typen.

Das Versenden einer Nachricht könnte also wie folgt aussehen:

```
ThreadId threadid;
Message sendmsg;
char *hello="Hello!";

sendmsg.id = 4;
sendmsg.msg.userMsg.p1 = hello;
tmMsgSend (threadid, &sendmsg);
```

Der einzige Nachrichtentyp, der für Benutzer-Threads vorgesehen ist, ist die Nachricht `userMsg` vom Typ `UserMessage`. Benutzer-Threads können also maximal drei Zeiger an einen anderen Benutzer-Thread versenden. Da alle Benutzer-Threads im gleichen Adressraum arbeiten, gibt es damit keine größeren Probleme. Problematisch wird es erst, wenn der sendende Thread beendet wird, da dann auch der allokierte Speicher freigegeben wird. Wenn der Empfänger dann auf die Daten zugreifen will, auf welche die empfangenen Zeiger verweisen, könnte es zu Problemen kommen. Wenn möglich, sollten also keine Zeiger verschickt werden. Um dieses Problem zu umgehen, könnte für die Daten eine geeignete Struktur erstellt und zum Verbund `SpecMessage` hinzugefügt werden. Die Informationen könnten dann in diese Struktur verpackt und gesendet werden.

Bei Erfolg liefert die Funktion `tmMsgSend` den Wert `TM_MSGSENDOK` zurück. Falls die Nachricht nicht gesendet werden konnte, weil der Ziel-Thread nicht existiert oder dessen Warteschlange bereits voll ist, wird der Wert `TM_MSGSENDFAILED` zurückgeliefert.

Zur Bestimmung der eigenen Thread-ID oder der des Empfängers kann die Funktion `tmGetInfo` verwendet werden. Diese Funktion liefert Informationen über einen Thread zurück und wird weiter unten genauer beschrieben.

4.2.2 Empfangen von Nachrichten

Nachrichten können mit der Funktion `tmMsgRecv` empfangen werden.

```
SyscallError tmMsgRecv (ThreadId* from,
                        MessageId msgId,
                        Message* msg,
                        int timeOut);
```

`from`: Thread, von dem eine Nachricht empfangen werden soll. Wenn dabei eine Thread-ID mit dem Wert `ANY` eingesetzt wird, wird eine Nachricht von einem beliebigen Sender empfangen, `*from` enthält dann die Thread-ID dieses Senders.

`msgId`: Typ der Nachricht, welchen man erwartet. Wenn an dieser Stelle der Wert `ANYMSGTYPE` eingesetzt wird, wird einfach die nächste Nachricht empfangen, egal um welchen Typ es sich handelt.

`msg`: Zeiger auf die empfangene Nachricht.

`timeOut`: Timeout[1] (in Millisekunden) für das Empfangen einer Nachricht, damit ein Thread nicht ewig wartet. Falls für `timeOut` der Wert `INFINITY` eingesetzt wird, blockiert sich der Thread solange, bis er eine Nachricht erhält, die den übergebenen Parametern entspricht.

Bei Erfolg liefert die Funktion `TM_MSGRECVOK` zurück, ansonsten `TM_MSGRECVFAILED`.

Falls der Thread, von dem eine Nachricht erwartet wird, nicht mehr existiert und daher keine Nachricht senden kann, liefert die Funktion auch `TM_MSGRECVFAILED` zurück.

Das nachfolgende Beispiel zeigt, wie eine Nachricht vom Thread `fromid` empfangen wird. Dabei blockiert sich der Thread solange, bis es eine entsprechende Nachricht gibt oder bis der Thread `fromid` beendet wird.

```
ThreadId fromid=ANY;
Message msg;
char *hello;

tmMsgRecv (&fromid, ANYMSGTYPE, &msg, INFINITY);
if(msg.id == 4)
  hello = msg.msg.userMsg.p1;
```

Anhand des Typs der Nachricht (`msg.id`) sollte der Empfänger erkennen, worum es sich handelt, um anschließend auf die richtige Struktur zuzugreifen. Die in der Datei Topsy/Messages.h definierten Nachrichten-Typen dienen nur zum Aufrufen der verschiedenen System-Calls, Benutzer-Threads können zur Kommunikation untereinander beliebige IDs verwenden.

[1]Die Angabe eines Timeouts bei dem MIPS R3000 Prozessor ist wirkungslos, die Threads warten also unendlich lange auf die Ankunft einer Nachricht.

4.3 Erzeugen von Threads

Threads können mit dem Befehl `tmStart` erzeugt werden.

```
SyscallError tmStart(ThreadId* id,
                     ThreadMainFunction mainFunction,
                     ThreadArg parameter,
                     char *name);
```

`id:` In dieser Variablen wird die ID des neu erstellten Threads gespeichert.

`mainFunction:` Zeiger auf die Hauptfunktion des neuen Threads.

`parameter:` Zeiger auf einen Speicherbereich, der dem neuen Thread als Parameter übergeben wird. Dieser Parameter vom Typ `ThreadArg` kann dabei beispielsweise einen Zeiger auf einen String speichern. Auch hier gibt es ähnliche Probleme wie beim Versenden von Nachrichten. Das Übergeben von Parametern geschieht in Form eines Zeigers auf einen zuvor allokierten Speicherbereich. Der Vater-Thread könnte terminiert werden bevor der neu erstellte Thread die Parameter auswertet. Die Daten in diesem Speicherbereich könnten dann allerdings bereits ungültig sein.

`name:` Name des Threads. Dieser Name kann beispielsweise dazu verwendet werden, die ID des Threads zu ermitteln.

Die Funktion `tmStart` liefert bei Erfolg `TM_STARTOK` und bei einem Fehler `TM_STARTFAILED` zurück.

```
void child() {
  ...
}

void parent() {
  ThreadId childid;
  tmStart (&childid, (ThreadMainFunction)child,
           (TreadArg)0, "child");
}
```

Beim Aufruf dieser Funktion `tmStart` wird eine Nachricht vom Typ `TM_START` mit den Daten des zu erzeugenden Threads an den Threadmanager weitergeschickt. Diese Nachricht wird von der Funktion `tmMain` empfangen und die Funktion `threadStart` wird mit den Daten des Threads aufgerufen.

```
typedef void* ThreadArg;
typedef void(*ThreadMainFunction)(ThreadArg);

ThreadId threadStart( ThreadMainFunction fctnAddr,
                      ThreadArg parameter,
                      AddressSpace space,
                      char* name,
                      ThreadId parentId,
                      ThreadPriority priority);
```

`fctnAddr`: Zeiger auf die Hauptfunktion des neuen Threads.

`parameter`: Zeiger auf die Parameter.

`space`: Adressraum, in welchem der Thread gestartet werden soll. In Topsy gibt es nur einen Kernadressraum und einen Benutzeradressraum. Andere Adressräume gibt es nicht. Beim Erzeugen eines Threads mit dem System-Call `tmStart` erhält der neue Thread den Adressraum des Vater-Threads.

`name`: Name des neuen Threads.

`parentId`: ID des Vater-Threads.

`priority`: Priorität des neuen Threads. In Topsy gibt es nur drei verschiedene Prioritäten: `KERNEL_PRIORITY`, `USER_PRIORITY` und `IDLE_PRIORITY`. Bei der Verwendung des System-Calls `tmStart` erbt der neue Thread die Priorität des Vater-Threads.

Die Funktion `threadStart` allokiert auf dem Heap Speicher zur Verwaltung der Informationen des Threads und im virtuellen Speicher wird ein 1024 Byte großer Stack für den neuen Thread allokiert. Auf dem Stack werden beispielsweise Rücksprungadressen abgelegt. Da die Größe des Stacks begrenzt ist und sich außerhalb dieses Stacks andere Daten befinden könnten, sollte ein Thread die Grenzen des Stacks nicht überschreiten. Es können also nicht beliebig viele Daten auf dem Stack abgelegt werden.

Der neue Thread benötigt auch eine ID. Falls keine IDs mehr vorhanden sind, werden die Ressourcen wieder freigegeben und die Funktion liefert einen Fehler zurück (`TM_THREADSTARTFAILED`). Eine Besonderheit von Topsy ist, dass Benutzer-Threads positive und Kern-Threads negative IDs erhalten.

Der nächste Schritt besteht darin, den Thread in die Hashliste und in die Threadliste einzutragen. Auch hier liefert die Funktion einen Fehler zurück, falls kein Speicher mehr vorhanden ist.

Für den neuen Thread müssen noch Strukturen angelegt werden. Das geschieht durch den Aufruf der Funktion `threadBuild` durch die Funktion `threadStart`.

```
void threadBuild( ThreadId id,
                  ThreadId parentId,
                  char* name,
                  ProcContext* contextPtr,
                  Address stackBaseAddress,
                  unsigned int stackSize,
                  ThreadMainFunction mainFunction,
                  ThreadArg parameter,
                  AddressSpace space,
                  Thread* threadPtr,
                  ThreadPriority priority);
```

`id`: ID des neuen Threads.

`parentId`: ID des Vater-Threads.

`name`: Name des Threads.

`contextPtr`: Zeiger auf eine Struktur vom Typ `ProcContext`. In dieser Struktur wird der Prozessor-Kontext des Threads gespeichert, beispielsweise der Stack-Zeiger oder die Register.

`stackBaseAddress`: Basisadresse des Stacks.

`stackSize`: Größe des Stacks.

`mainFunction`: Zeiger auf die Hauptfunktion.

`parameter`: Zeiger auf die Parameter.

`space`: Adressraum des Threads.

`threadPtr`: Zeiger auf eine Struktur vom Typ `Thread`. In dieser Struktur trägt die Funktion `threadBuild` die ihr übergebenen Daten ein.

`priority`: Priorität des Threads.

```
typedef struct Thread_t {
    ProcContextPtr contextPtr;
    ThreadId id;
    char name[MAXNAMESIZE];
    ThreadId parentId;
    Address stackStart;
    Address stackEnd;
    MessageQueue msgQueue;
    SchedulerInfo schedInfo;
    ThreadStatistics stat;
```

```
} Thread;
```

Die Variable `schedInfo` in dieser Struktur enthält den Zustand und die Priorität des Threads. In `stat` werden die CPU-Zyklen gezählt, die der Thread verbraucht hat. Damit könnte etwa ein Multilevel-Feedback-Scheduler (MLFB) implementiert werden.

Die Funktion `threadBuild` sorgt auch dafür, dass der neue Thread in den Scheduler eingetragen wird und dass der Programmzähler, der Stack-Pointer und die Register des neuen Threads initialisiert werden. Außerdem wird der Stack des Threads initialisiert. Auf den Stack wird eine Sprungadresse abgelegt, welche zur Funktion `automaticThreadExit` springt, falls der Thread seine Hauptfunktion verlässt. Damit erkennt das Betriebssystem, dass ein Thread beendet wurde.

Nachdem die Funktion `threadBuild` zurückgekehrt ist, setzt die Funktion `threadStart` den Status des Threads auf READY (siehe Abbildung 4.1 auf Seite 11). Das signalisiert dem Scheduler, dass dieser Thread jetzt dem Prozessor zugeordnet und somit abgearbeitet werden kann.

Bei Erfolg liefert die Funktion `threadStart` die ID des neu erzeugten Threads zurück, ansonsten `TM_THREADSTARTFAILED`.

Abbildung 4.4 soll noch einmal verdeutlichen, welche Schritte beim Erzeugen eines neuen Threads durchgeführt werden.

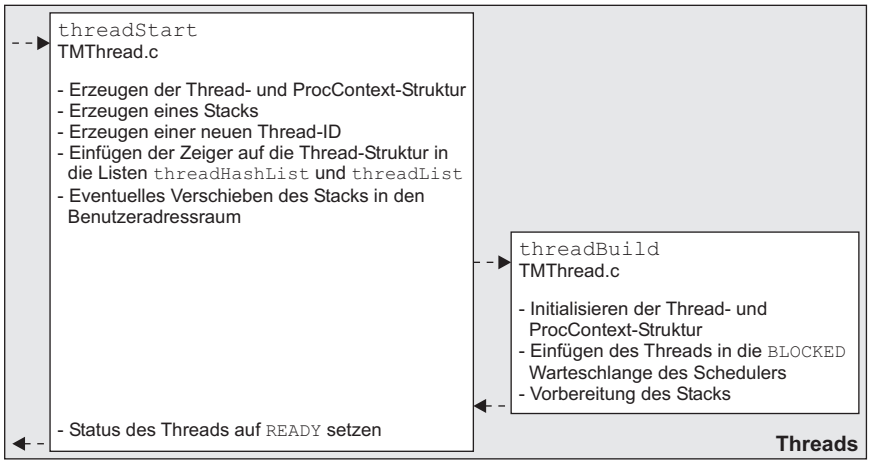

Abbildung 4.4: Erzeugung eines neuen Threads

Ein Thread kann seine eigene Ausführung zu Gunsten der anderen Threads anhalten. Dies geschieht durch den Aufruf der Funktion `tmYield`.

4.4 Beenden von Threads

Ein Thread wird beendet, wenn er aus seiner Hauptfunktion zurückkehrt (z. B. durch `return`) oder wenn er explizit die Funktion `tmExit` aufruft.

Wenn ein Thread seine Hauptfunktion verlässt, befinden sich noch immer einige Daten auf dem Stack, welche beim Erzeugen des Threads dort abgelegt wurden. Das oberste Datum ist eine Sprungadresse zur Funktion `automaticThreadExit`. Das letzte Datum ist ein Verweis auf die Nachricht `threadExitMsg`. Die Funktion `automaticThreadExit` sorgt dafür, dass diese Nachricht an den Threadmanager gesendet wird. Damit weiß der Threadmanager, dass ein Thread beendet wurde.

Ein Thread kann jedoch auch gezielt terminiert werden. Mit dem Befehl `tmKill` kann ein Thread einen anderen beenden, wenn dessen ID bekannt ist. Ein Benutzer-Thread darf jedoch keine Kern-Threads terminieren, ansonsten wird er selbst beendet.

Beim Beenden eines Threads wird die Funktion `threadDestroy` aufgerufen. Diese Funktion entfernt den Thread aus dem Scheduler und gibt den Speicher wieder frei, der für den Thread-Deskriptor und den Stack allokiert wurde. Die Funktion gibt auch den Speicher frei, den der Thread allokiert hatte. Außerdem weckt die Funktion die Threads, die auf eine Nachricht des terminierten Threads warten.

4.5 Informationen über Threads

Manchmal ist es nötig, dass ein Thread einige Informationen über sich selbst und andere Threads erfährt, beispielsweise, wenn er an jemanden eine Nachricht senden oder einen anderen Thread terminieren will.

Dazu gibt es mehrere Funktionen:

```
SyscallError tmGetInfo(ThreadId about, ThreadId* tid,
                       ThreadId* parentTid);
SyscallError tmGetFirst(ThreadInfo* info);
SyscallError tmGetNext(ThreadInfo* info);
SyscallError tmThreadByName(char *name, ThreadID* tid);
SyscallError tmGetIDInfo(ThreadID about, ThreadInfo* info);
```

Mit `tmGetInfo` können Informationen über einen bestimmten Thread `about` abgerufen werden. Die ID des Threads und die ID des Vater-Threads werden in den Variablen `tid` und `parentTid` zurückgeliefert. Wenn ein Thread Informationen über sich selbst abrufen will und seine ID nicht kennt, dann kann er für den Parameter `about` den Wert `SELF` einsetzen.

Mit den beiden Funktionen `tmGetFirst` und `tmGetNext` können alle aktiven Threads durchlaufen werden. Die Informationen über diese Threads werden in einer Struktur vom Typ `ThreadInfo` gespeichert.

```
typedef struct ThreadInfo_t {
    ThreadId tid;
    ThreadId ptid;
    ThreadInfoStatus status;
    char name[MAXNAMESIZE];
} ThreadInfo;
```

Diese Struktur enthält die ID des Threads, die ID seines Vater-Threads, den Status und den Namen des Threads. Der Status des Threads kann die Werte `Info_RUNNING`, `Info_READY` oder `Info_BLOCKED` annehmen.

Die ID eines Threads kann auch mit `tmThreadByName` über seinen Namen ermittelt werden. Informationen zu einer bestimmten Thread-ID erhält man durch Aufruf der Funktion `tmGetIDInfo`. Diese Funktion liefert ein Element vom Typ `ThreadInfo` zurück.

All diese Funktionen liefern bei Erfolg den Wert `TM_INFOOK` zurück, ansonsten `TM_INFOFAILED`.

Die Aufrufe dieser Funktionen werden weitergeleitet an die Funktion `tmMain`, die diese Nachrichten vom Typ `TM_INFO` empfängt, bearbeitet und zurückschickt.

4.6 Scheduler

Der Topsy-Scheduler ist ein einfacher Round-Robin-Scheduler mit Prioritäten. Abbildung 4.5 zeigt eine Skizze dieses Schedulers. Der Scheduler hat die Aufgabe, neu ankommende Aufträge (Threads) je nach ihrer Priorität in eine bestimmte Warteschlange einzuordnen. Für jede Priorität gibt es zwei Warteschlangen bzw. Listen. Eine Liste enthält die lauffähigen Threads, deren Status READY ist und die zweite Liste enthält blockierte Threads.

Weitere Informationen über Round-Robin Scheduling findet man in [Tan90a, Abschnitt 2.4.1].

Topsy kennt drei verschiedene Prioritäten: `KERNEL_PRIORITY`, `USER_PRIORITY` und `IDLE_PRIORITY`.

Die Funktion `schedule` hat die Aufgabe, einen Thread auszuwählen, der als nächstes abgearbeitet wird. Der Idle-Thread garantiert, dass immer ein Thread

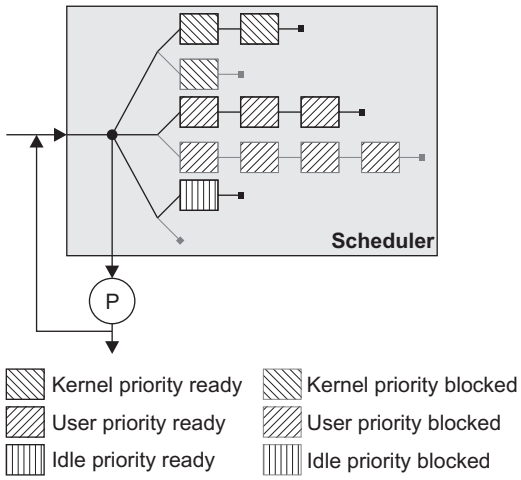

Abbildung 4.5: Scheduler

existiert, der zugeordnet werden kann. Bevor jedoch ein neuer Thread ausgewählt wird, wird der zuvor bearbeitete ans Ende seiner Warteschlange zurückgeschrieben, falls er mit seiner Bearbeitung noch nicht fertig ist. Um einen neuen, geeigneten Thread zu finden, werden alle Warteschlangen durchsucht, beginnend bei der mit der höchsten Priorität. Der erste Thread mit dem Status READY wird ausgewählt und seine ID wird nach `scheduler.running` geschrieben. Die Funktion `schedule` wird von der Funktion `tmClockHandler` (siehe Abschnitt 4.8) aufgerufen – dem Interrupt-Handler für die Uhr (siehe Abbildung 4.6).

Abbildung 4.6: Aufruf des Schedulers

Aufträge, die nach ihrem Zeitquantum (10 Millisekunden) noch nicht fertig sind, werden wieder in die Warteschlange eingeordnet.

Der Nachteil dieses Schedulers ist, dass Threads mit einer niedrigen Priorität erst dann ausgeführt werden, wenn keine höherprioren Threads auf eine Abarbeitung warten.

Der Scheduler wird durch die Funktion `schedulerInit` initialisiert. Diese Funktion initialisiert die Warteschlangen. Zunächst befinden sich noch keine Threads im System, also sind auch die Warteschlangen leer. Für jede Priorität gibt es zwei Listen, eine Liste verwaltet die wartenden Prozesse (Status READY), die andere die blockierten Prozesse (Status BLOCKED).

```
typedef struct SchedPriority_t {
    List ready;
    List blocked;
} SchedPriority;

typedef struct Scheduler_t {
    ThreadPtr running;
    SchedPriority prioList[NBPRIORITYLEVELS];
} Scheduler;
```

Die Variable `scheduler.running` gibt an, welcher Thread momentan bearbeitet wird, `prioList` enthält für jede Priorität eine Liste mit wartenden und eine mit blockierten Prozessen.

Es gibt insgesamt `NBPRIORITYLEVELS` Prioritäten, wobei 0 die höchste Priorität darstellt.

Mit der Funktion `schedulerInsert` wird ein neuer Thread in die Warteschlange eingefügt. Der Status des neuen Threads wird zuerst auf BLOCKED gesetzt. Dann wird der Thread an den Anfang der Warteschlange eingefügt.

```
void schedulerInsert(Thread* threadPtr,
                     ThreadPriority priority);
```

Die Funktion `schedulerRemove` entfernt einen Thread aus den Warteschlangen. Je nachdem, ob der Thread bereit oder blockiert ist, wird er aus der jeweiligen Liste entfernt.

```
void schedulerRemove(Thread* threadPtr);
```

Mit den Funktionen `schedulerSetReady` und `schedulerSetBlocked` kann der Status des Threads festgelegt werden. Beide Funktionen erwarten als Argument einen Zeiger auf die Thread-Struktur (siehe Abschnitt 4.3).

4.7 Interrupts und Exceptions

Das Modul Threads bietet die Möglichkeit, Interrupt- und Exception-Handler zu installieren, die auf bestimmte Ereignisse reagieren. Eine Exception tritt auf, wenn im System ein Fehler auftritt, beispielsweise eine Division durch null oder einen Überlauf. Interrupts werden von Geräten (Hardware-Interrupts) oder von Software (Software-Interrupts) ausgelöst. Exceptions und Interrupts haben zur

Folge, dass das System angehalten und der entsprechende Handler aufgerufen wird.

Mit den Funktionen `tmSetInterruptHandler` und `tmSetExceptionHandler`, welche in der Datei Threads.h exportiert werden, können solche Handler installiert werden.

```
typedef void (*ExceptionHandler)(ThreadId id);
typedef void (*InterruptHandler)(void* arg);

InterruptHandler tmSetInterruptHandler( InterruptId id,
                    InterruptHandler intHdler,
                    void* arg);
```

`id`: Nummer des Interrupts. In Topsy gibt es 6 Hardware-Interrupts (Nummer 0 bis 5) und 2 Software-Interrupts (6, 7).

`intHdler`: Zeiger auf den Interrupt-Handler.

`arg`: Zeiger auf eventuelle Argumente.

Mit der Funktion `tmSetExceptionHandler` können Handler installiert werden, welche bei einem Ausnahmefehler aufgerufen werden.

```
ExceptionHandler tmSetExceptionHandler( ExceptionId id,
                    ExceptionHandler excHdler);
```

`id`: Nummer der Exception. Es gibt insgesamt 32 verschiedene Exceptions.

`excHdler`: Zeiger auf den Exception-Handler.

Dem Exception-Handler wird als einziges Argument die ID des Threads übergeben, welcher den Fehler verursacht hat.

In der Funktion `initBasicExceptions`, Datei Threads/TMInit.c, werden für alle Interrupts und Exceptions Dummy-Handler installiert. Die Funktion `tmInit` installiert dann die eigentlichen Exception-Handler (z. B. für Overflow-Exceptions) durch den Aufruf von `tmInstallErrorHandlers`, Datei Threads/mips/TMError.c. In dieser Funktion wird auch ein Interrupt-Handler installiert, nämlich der Interrupt-Handler für die Uhr (`tmClockHandler`).

Die Funktionen zum Setzen von Interrupt- bzw. Exception-Handlern sind hardwareabhängig und deshalb jeweils für die unterschiedlichen Architekturen implementiert. Für einen Interrupt oder eine Exception kann immer nur ein Handler installiert werden.

Um einen Handler zu deinstallieren, genügt ein Aufruf der entsprechenden Funktion mit dem Parameter `NULL`. Das folgende Beispiel zeigt, wie der Exception-Handler für die Exception mit der Nummer 2 entfernt wird:

```
tmSetExceptionHandler(2,NULL);
```

4.8 Uhrzeit

Jedesmal, wenn die Uhr des Prozessors einen Hardware-Interrupt auslöst, wird der dazugehörige Interrupt-Handler (`tmClockHandler`) aufgerufen und die Zeit wird um einen Tick erhöht. Der Benutzer hat die Möglichkeit, mit den Funktionen `tmGetTime` die Zeit abzurufen und mit `tmSetTime` zu setzen.

```
SyscallError tmGetTime(unsigned long* seconds,
                       unsigned long* microSeconds);
SyscallError tmSetTime(unsigned long seconds,
                       unsigned long microSeconds);
```

Die Funktion `tmClockHandler` (Threads/TMScheduler.c) sorgt auch dafür, dass der Scheduler einen neuen Thread auswählt, der als nächstes ausgeführt wird. Zu diesem Zweck wird die Funktion `schedule` (siehe Abschnitt 4.6) aufgerufen.

Kapitel 5

Das Modul ThreadsLight

Zwischen den schwergewichtigen und den leichtgewichtigen Threads gibt es einige Unterschiede.

Ein Unterschied besteht darin, dass die leichtgewichtigen Threads keine Nachrichten zwischen verschiedenen Adressräumen austauschen können und die Nachrichten werden nicht in einer Warteschlange zwischengespeichert. Außerdem muss bereits vorher bekannt sein, wieviele Threads benötigt werden.

Einen weiteren Unterschied gibt es beim Scheduling. Wenn einer dieser Threads gestartet wird, läuft er solange, bis er beendet ist. Es gibt also kein Zeitquantum und keinen Scheduler für diese Threads.

Der Vorteil dieser Threads liegt darin, dass sie schnell sind und wenig Speicherplatz beanspruchen.

Es sieht allerdings so aus, als befände sich dieses Modul noch in der Entwicklungsphase, da es bisher noch keine Beispiele zu diesen Threads gibt und dieses Modul auch nicht compiliert wird.

Kapitel 6

Das Modul IO

Das Modul IO enthält verschiedene Gerätetreiber und bietet auch die Möglichkeit, neue Treiber hinzuzufügen.

Module werden fest in den Kern eincompiliert, ein dynamisches Nachladen von Modulen ist in der aktuellen Version von Topsy noch nicht möglich. [Bre99] befasst sich mit dem dynamischen Laden von Modulen.

Benutzerprogramme können über System-Calls Geräte öffnen, Daten schreiben und lesen und das Gerät am Ende wieder schließen. Somit wäre es möglich, ein Dateisystem als Treiber zu implementieren. Das Dateisystem könnte geöffnet werden, Befehle entgegennehmen und diese abarbeiten.

Die verschiedenen Treiber laufen als Threads im Kernadressraum und warten dort auf Anfragen, um diese zu bearbeiten.

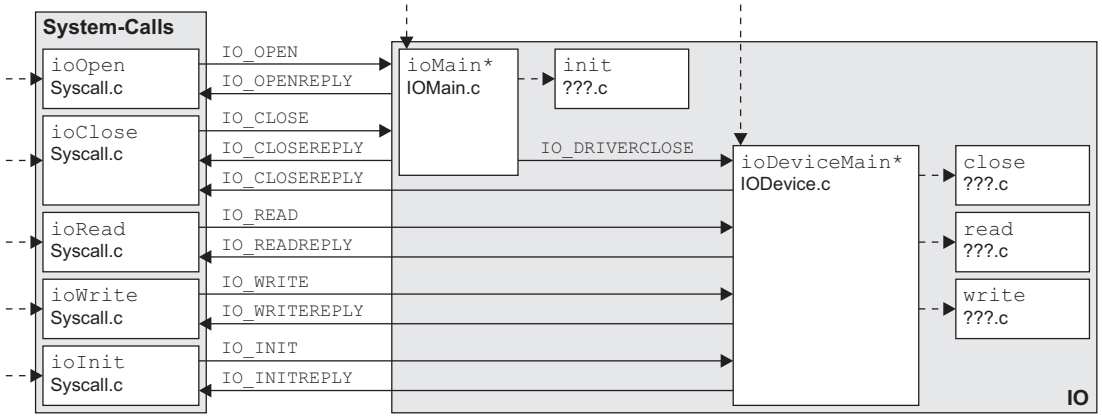

Abbildung 6.1: Überblick über das Modul IO

Abbildung 6.1 gibt einen Überblick über das Modul IO und zeigt, welche System-Calls es zur Arbeit mit Treibern gibt und was geschieht, wenn ein solcher System-

Call aufgerufen wird. Die Funktionen `init`, `open`, `close`, `read` und `write` sind dabei Treiber-spezifisch.

6.1 Vorhandene Gerätetreiber

Topsy enthält für den MIPS Prozessor bereits einige Treiber.

Es gibt einen Treiber für die seriellen Schnittstellen (`ttya` und `ttyb`). Die Funktionen für diesen Treiber sind in der Datei IO/Drivers/SCN2681_DUART.c programmiert. Jedesmal, wenn Daten empfangen werden, wird ein Interrupt ausgelöst und der Interrupt-Handler des Treibers liest die Daten von der seriellen Schnittstelle und speichert sie in einem Ringpuffer. Dieser Ringpuffer kann dann von anderen Threads ausgelesen werden.

Die serielle Schnittstelle `ttyb` ist gleichzeitig auch die Konsole. Auf der Konsole können Kern-Threads mit folgenden Funktionen Strings und Zahlen ausgeben:

```
void ioConsolePutString(const char* s);
void ioConsolePutHexInt(int x);
void ioConsolePutInt(int x);
```

Benutzerprogramme müssen die Konsole über den Treiber ansprechen. Um Text auf der Konsole auszugeben, muss dieser mit dem Befehl `ioWrite` an die Konsole geschickt werden. Benutzereingaben können mit `ioRead` eingelesen werden.

Auf dem MIPS Prozessor bzw. dem Evaluation Board befinden sich noch weitere Bausteine, für welche Topsy Treiber bereitstellt. Topsy bietet Treiber für den Hardware-Timer und das FPGA (Field Programmable Gate Array).

Ein Loopback-Device (IO/Drivers/Loopback.c) kann als Ringpuffer verwendet werden. Der Treiber ist so konfiguriert, dass er 512 Bytes aufnehmen kann.

6.2 Initialisierung

Der IO-Manager `ioMain` wird als Thread von der Funktion `tmMain` gestartet. In der Funktion `ioMain` werden zunächst wieder die Datenstrukturen zur Verwaltung der verschiedenen Gerätetreiber initialisiert, alle Treiber werden initialisiert und als Threads gestartet. Die Funktion `ioMain` läuft anschließend in eine Endlosschleife und wartet auf `IO_OPEN`- und `IO_CLOSE`-Nachrichten.

6.3 Verwaltung

Die Informationen über die verfügbaren Treiber werden in `ioDeviceTable` gehalten. Diese Variable ist ein Array mit `IODeviceDesc`-Elementen.

```
typedef struct IODeviceDesc_t {
        char* name;
        Address base;
        char* buffer;
        InterruptId interrupt;
        InterruptHandler interruptHandler;
        ThreadId thread_id;
        ReadFunction read;
        WriteFunction write;
        CloseFunction close;
        InitFunction init;
        MessageHandler handleMsg;
        Boolean isInitialised;
        void* extension;
} IODeviceDesc;
```

`name:` Name des Treibers. Der Treiber wird als Thread gestartet, welcher auch diesen Namen erhält.

`base:` Basisadresse des Treibers. Um ein Gerät (beispielsweise die serielle Schnittstelle) zu steuern, muss der Treiber mit ihm kommunizieren. Dazu wird ein Speicherbereich benötigt, über welchen dieses Gerät angesprochen werden kann. Die Basisadresse dieses Speicherbereichs wird in der Variablen `base` gespeichert.

`buffer:` Puffer, in welchem der Treiber Daten ablegen kann. Ein Beispiel hierfür ist wieder die serielle Schnittstelle bzw. die Konsole. Der Benutzer kann über die Tastatur Daten in die Konsole eingeben. Jedesmal, wenn die serielle Schnittstelle Daten erhält, wird ein Interrupt ausgelöst und der Treiber muss die Zeichen in einen Puffer schreiben, der irgendwann von einem Benutzerprogramm ausgelesen wird, beispielsweise von der Shell.

`interrupt:` Nummer des Interrupts. Wie bereits oben erwähnt wurde, muss der Treiber darauf reagieren, wenn das Gerät etwas von ihm will. Der Interrupt-Handler für die serielle Schnittstelle reagiert beispielsweise auf den Interrupt 5. Wenn der Treiber keinen Interrupt verwendet, dann muss hier -1 angegeben werden.

`interruptHandler:` Interrupt-Handler des Treibers.

`thread_id`: Thread-ID des Treiber-Threads. Jeder Treiber wird als Thread gestartet und erhält eine ID, welche nach dem Start des Treibers hier eingetragen wird.

`read`: Zeiger auf eine Funktion zum Lesen von Daten.

`write`: Zeiger auf eine Funktion zum Schreiben von Daten.

`close`: Zeiger auf eine Funktion zum Schließen des Treibers.

`init`: Zeiger auf eine Funktion zum Initialisieren des Treibers.

`handleMsg`: Zeiger auf den Message-Handler des Treibers. Da ein Treiber auch als Thread gestartet wird, können andere Threads Nachrichten an ihn schicken. Erhält ein Treiber eine Nachricht, dann wird dieser Message-Handler aufgerufen.

`isInitialised`: Dieses Flag gibt an, ob der Treiber bzw. das Gerät initialisiert wurde. Dieses Flag sollte der Treiber setzen, wenn seine Initialisierungsroutine aufgerufen und erfolgreich bearbeitet wurde.

`extension`: Diese Erweiterung kann der Treiber nutzen, um weitere Daten zu speichern.

6.4 System-Calls

6.4.1 Öffnen

Der System-Call `ioOpen` veranlasst das Senden einer entsprechenden Nachricht an den IO-Manager. Dieser sucht in einer Hash-Liste nach dem Treiber und liefert bei Erfolg die Thread-ID des geöffneten Treibers zuück.

```
SyscallError ioOpen(int deviceNumber, ThreadId* id);
```

`deviceNumber`: Nummer des Treibers. Jeder Treiber hat eine eindeutige Nummer. Diese Nummern werden in der Datei IO/mips/IOArch.h definiert. Beim Erstellen eines Treibers ist darauf zu achten, dass hier keine Lücken entstehen. Es darf also nicht vorkommen, dass es einen Treiber mit der Nummer 0 und einen mit der Nummer 2 gibt, aber keinen Treiber mit der Nummer 1. Über diese Nummern wird auf die Struktur `ioDeviceTable` zugegriffen, welche die Daten über alle Treiber enthält. Es ist also auch darauf zu achten, dass die Reihenfolge der Einträge dieser Struktur mit der Reihenfolge der Treiber-Nummern übereinstimmt.

`id`: In dieser Variablen wird die Thread-ID des Treibers zurückgeliefert, falls die Funktion erfolgreich war.

Bei einem Fehler liefert die Funktion `ioOpen` den Wert `IO_OPENFAILED` zurück, bei Erfolg den Wert `IO_OPENOK`.

Der Befehl `ioOpen` dient nur dazu, die Thread-ID des Treibers zu ermitteln. Auf den Treiber selbst hat dieser Befehl keine Auswirkung. Der Treiber wird bei einem Open-Aufruf nicht gestartet. Das Starten der einzelnen Treiber übernimmt der IO-Manager beim Systemstart.

6.4.2 Schließen

Treiber können mit dem System-Call `ioClose` wieder geschlossen werden. Dazu wird wieder eine Nachricht an den IO-Manager geschickt, der eine entsprechende Nachricht mit einer Close-Aufforderung an den Treiber weiterschickt. Der Treiber sendet daraufhin eine Antwort an den System-Caller zurück.

```
SyscallError ioClose(ThreadId id);
```

Im Fehlerfall liefert die Funktion `IO_CLOSEFAILED` zurück, ansonsten `IO_CLOSEOK`.

Der Close-Aufruf dient nur dazu, die `close`-Prozedur des Treibers auszuführen. Der Treiber selbst wird jedoch nicht beendet, sondern er läuft als Thread weiter und wartet auf neue Befehle.

Im Normalfall sendet der Treibers selbst eine `IO_CLOSEREPLY`-Nachricht an den Caller zurück (siehe Abbildung 6.1 auf Seite 29). Falls jedoch bereits vorher das Senden der Nachricht `IO_DRIVERCLOSE` scheitert, sendet der IO-Manager eine Antwort an den Caller mit einer entsprechenden Fehlermeldung.

6.4.3 Initialisieren

Der Treiber kann mit dem System-Call `ioInit` initialisiert werden. Dabei wird die Initialisierungs-Routine des Treibers ausgeführt.

Der System-Call liefert bei Erfolg `IO_INITOK` zurück, ansonsten `IO_INITFAILED`.

6.4.4 Lesen und Schreiben

Mit den Befehlen `ioRead` und `ioWrite` kann der Treiber dazu veranlasst werden, Daten zu lesen oder zu schreiben. Diese beiden System-Calls werden nicht wie die Open- und Close-Aufrufe an den IO-Manager weitergeleitet, sondern direkt an den Treiber. Der Treiber-Thread ruft dann die Funktion zum Lesen bzw. Schreiben auf (`ReadFunction read`, `WriteFunction write`) und liefert deren Ergebnis zurück.

```
SyscallError ioRead(ThreadId id, char* buffer,
                    unsigned long int* nOfBytes);

SyscallError ioWrite(ThreadId id, char* buffer,
                     unsigned long int* nOfBytes);
```

`id:` Thread-ID des Treibers. Diese ID kann mit dem System-Call `ioOpen` ermittelt werden.

`buffer:` Puffer der Daten enthält bzw. Puffer, in welchen der Treiber die Daten schreiben soll.

`nOfBytes:` Anzahl der Bytes, die gelesen bzw. geschrieben werden sollen. Diese Variable enthält nach Rückkehr die Anzahl der tatsächlich gelesenen bzw. geschriebenen Bytes.

Im Fehlerfall liefert die Funktion `ioWrite IO_WRITEFAILED` zurück, ansonsten `IO_WRITEOK`.

Die Funktion `ioRead` liefert `IO_READFAILED` bei einem Fehler zurück, bei Erfolg den Wert `IO_READOK`.

6.5 Hinzufügen eigener Treiber

6.5.1 Anpassen der verschiedenen Dateien

Um einen eigenen Treiber hinzuzufügen, müssen Änderungen in verschiedenen Dateien durchgeführt werden.

In der Datei IO/mips/IOArch.h muss der Wert `IO_DEVCOUNT` erhöht werden und es muss eine Nummer für den neuen Treiber hinzugefügt werden (`device numbers`).

In IO/mips/IODevTable.h werden die Informationen über den neuen Treiber eingetragen. Hier muss auch die Datei, in welche der Treiber seine Funktionen exportiert, eingebunden werden. Zur Struktur `IODeviceDesc` müssen die Daten (Name, Interrupt, Adresse, Funktionen, ...) des neuen Treibers hinzugefügt werden (siehe Abschnitt 6.3).

6.5.2 Anpassen des Makefiles

Damit der neue Treiber bei einem späteren Compilerlauf auch mitcompiliert wird, muss er noch in das entsprechende Makefile (z. B. Makefiles/Makefile.IDT.mips) eingetragen werden. In diesem Makefile muss lediglich die Variable `IOFILES-DRIVER` um die neue Datei erweitert werden.

6.5.3 Programmieren des Treibers

Die Schnittstelle zwischen dem eigentlichen Treiber und dem Kern bilden die Dateien IO/IODevice.c und IO/IODevice.h. Nach dem Starten des Treibers läuft die Routine `ioDeviceMain` in eine Endlosschleife und leitet READ-, WRITE-, CLOSE- und INIT-Befehle an den Treiber weiter (siehe Abbildung 6.1 auf Seite 29).

Um also den Treiber zu programmieren, genügt es, die entsprechenden Funktionen zu implementieren.

```
typedef Error(*ReadFunction)(IODevice this, ThreadId, char*,
                             long int*);
typedef Error(*WriteFunction)(IODevice this, ThreadId, char*,
typedef Error(*CloseFunction)(IODevice this);
typedef Error(*InitFunction)(IODevice this);
typedef void(*MessageHandler)(IODevice this, Message*);
```

Es können also Funktionen zum Lesen, Schreiben, Schließen und Initialisieren des Treibers programmiert werden sowie ein Message-Handler, der die Nachrichten bearbeitet, die an den Treiber geschickt werden.

Zusätzlich besteht die Möglichkeit, einen Interrupt-Handler hinzuzufügen, falls ein Gerätetreiber auf Interrupts des Gerätes reagieren muss/soll.

Das folgende Beispiel zeigt einen einfachen Treiber, der einen Speicherbereich mit einer bestimmten Anzahl von Nullen füllt. Der Treiber arbeitet also ähnlich wie /dev/zero bei UNIX-Systemen. Die einzige Funktion, die etwas macht, ist `devZero_read`. Diese Funktion füllt den übergebenen Speicherbereich mit Nullen. Die anderen Funktionen sind nur dazu da, um "positive" Werte zurückzugeben, damit es an anderen Stellen nicht zu Fehlermeldungen kommt.

```
#include "Drivers/zero.h"
#include "Support.h"
```

```
Error devZero_init(IODevice this) {
  return IO_INITOK;
}

Error devZero_read(IODevice this, ThreadId threadId,
  char* buffer, long int* size)
{
  /* Speicherbereich mit Nullen fuellen */
  zeroOut (buffer, *size);
  return IO_READOK;
}

Error devZero_write(IODevice this, ThreadId threadId,
  char* buffer, long int* size)
{
  return IO_WRITEOK;
}

Error devZero_close(IODevice this) {
  return IO_CLOSEOK;
}
```

Ein Programm, welches diesen Treiber verwendet, könnte beispielsweise so aussehen:

```
#include "../Topsy/Syscall.h"

void ZeroTest () {
  ThreadId zero;
  unsigned long int size=10;
  char buffer[10];
  ...
  ioOpen (IO_ZERO, &zero);
  ioRead (zero, buffer, &size);
  ...
  ioClose (zero);
}
```

Kapitel 7

Das Modul Memory

Das Modul Memory ist für die Verwaltung des Speichers zuständig. Dazu gehört die Initialisierung, Bereitstellung und Freigabe des Speichers sowie eine Zuordnung von virtuellen zu physikalischen Adressen.

Abbildung 7.1 gibt einen Überblick über die Struktur des Moduls sowie über die vorhandenen System-Calls, die benutzt werden können, um beispielsweise Speicher zu allokieren oder wieder freizugeben.

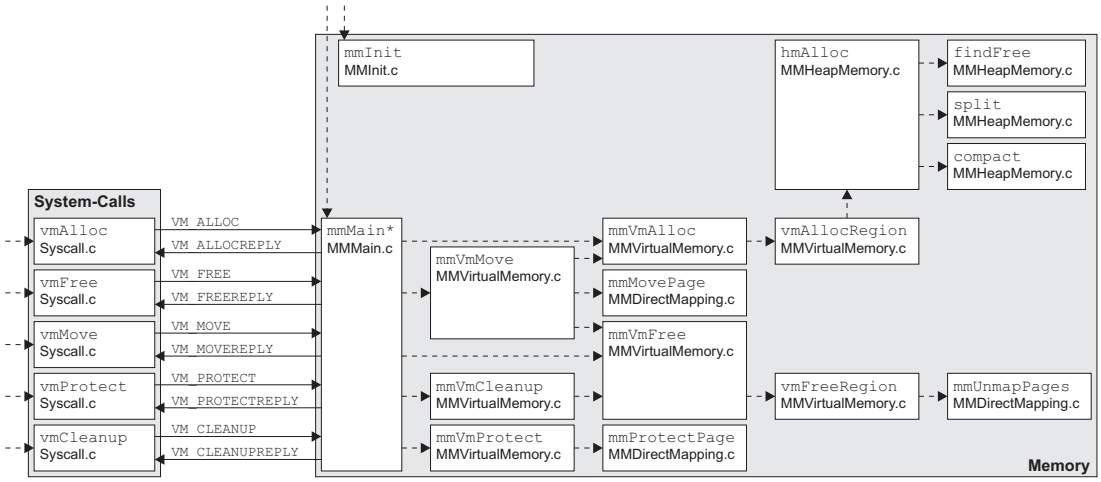

Abbildung 7.1: Überblick über das Modul Memory

7.1 Initialisierung

Beim Starten von Topsy wird die Funktion `mmInit` aufgerufen. Diese Funktion installiert zunächst die Fehlerbehandlungsroutinen (`mmInstallErrorHandlers`),

initialisiert das Mapping, erzeugt den Kern-Heap und sorgt durch einen Aufruf von `mmVmInit` dafür, dass der virtuelle Speicher initialisiert wird.

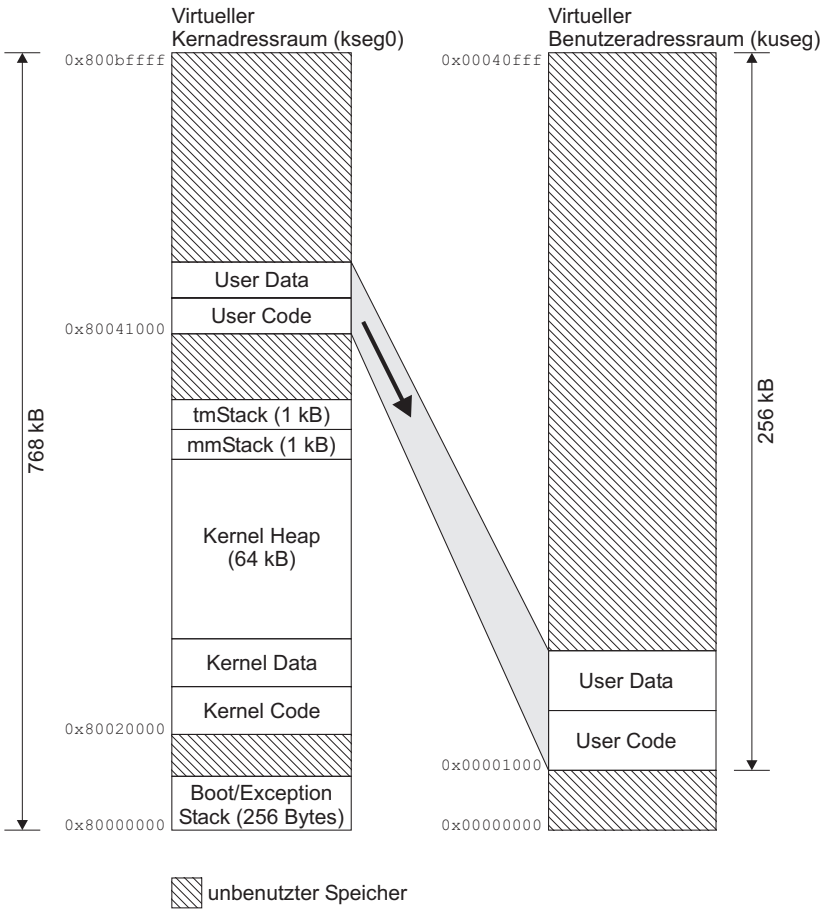

Abbildung 7.2: Speicher-Layout

Abbildung 7.2 zeigt das Speicher-Layout des virtuellen Kern- sowie Benutzeradressraumes. Gleich nach dem Bootvorgang befinden sich auch der Benutzer-Code und die Benutzer-Daten im Kernadressraum an Adresse `0x80041000`. Die Funktion `mmInitMemoryMapping` sorgt dafür, dass die Zuordnung von virtuellen zu physikalischen Adressen initialisiert und die Benutzer-Teile in den virtuellen Benutzeradressraum kopiert werden. Im virtuellen Benutzeradressraum bleibt die unterste Seite unbenutzt, um ungültige `NULL`-Zeiger abzufangen.

7.2 Virtueller Speicher

Der virtuelle Speicher wird in Seiten (pages) einer bestimmten Größe (bei Topsy 256 Bytes) aufgeteilt. Der physikalische Speicher wird in sogenannte Kacheln

(frames) aufgeteilt (beim MIPS R3000 je 4 kB). Die unterschiedlichen Größen der virtuellen und physikalischen Seiten sind nur zulässig, da bei Topsy der virtuelle Speicher direkt auf den physikalischen Speicher abgebildet wird (der für den Benutzeradressraum benötigte TLB wird nur einmal initialisiert und dann nicht mehr geändert). Bei einem normalen Mapping müssten virtuelle und physikalische Seiten gleich groß sein.

Da der virtuelle Adressraum größer als der physikalische Adressraum ist, ist keine direkte Zuordnung von virtuellen zu physikalischen Adressen möglich. Um also zu einer virtuellen eine physikalische Adresse zu erhalten, müssen diese Informationen in einer Seiten-Kachel-Tabelle gespeichert werden. Da das Durchsuchen einer solchen Tabelle sehr aufwendig ist, bieten die meisten Prozessoren einen Hardware-Cache, den Translation Lookaside Buffer (siehe Abschnitt 7.3). Dieser Puffer kann mehrere Zuordungen von virtuellen zu physikalischen Seiten speichern, damit nicht bei jedem Speicherzugriff die Seiten-Kachel-Tabelle durchsucht werden muss.

Einen guten Überblick über virtuellen Speicher gibt [Tan99, Abschnitt 6.1].

7.2.1 Initialisierung des virtuellen Speichers

Topsy unterteilt den virtuellen Adressraum in einen Benutzer- und einen Kernadressraum. Abbildung 7.3 zeigt, in welche Segmente der virtuelle Adressraum beim MIPS R3000 Prozessor unterteilt ist und wie diese Segmente dem physikalischen Speicher zugeordnet werden.

Der virtuelle Speicher ist in die Segmente kuseg, kseg0, kseg1 und kseg2 unterteilt, wobei Topsy nur die Segmente kseg0 und kuseg benutzt.

Die Segmente kuseg sowie kseg2 können beliebig abgebildet werden. Dabei werden Hilfsmittel benötigt, um zu einer virtuellen Adresse die zugehörige physikalische Adresse zu finden (TLB, Tabellen). Da der TLB beim MIPS R3000 nur 64 Einträge speichern kann und es bei Topsy keine Seiten-Kachel-Tabelle gibt, ist der Benutzeradressraum auf 256 kB beschränkt. Alle Benutzer-Threads müssen sich also diese 256 kB teilen.

Die Segmente kseg0 und kseg1 (jeweils 512 MB) werden dabei direkt auf die unteren 512 MB des physikalischen Speichers abgebildet. Der Vorteil einer direkten Abbildung besteht darin, dass kein Zugriff auf den TLB oder eine Seiten-Kachel-Tabelle benötigt wird, um die Adressen umzusetzen. Aus diesem Grund gibt es bei Topsy auch keine Beschränkung des Kernadressraums auf 256 kB.

Der Kernadressraum wird weiter in verschiedene Regionen aufgeteilt. Die Aufteilung des Speichers wird durch die Funktion `mmVmInit` durchgeführt. Es werden Regionen für den Boot/Exception Stack, Kern-Code, Kern-Daten, Kern-Heap,

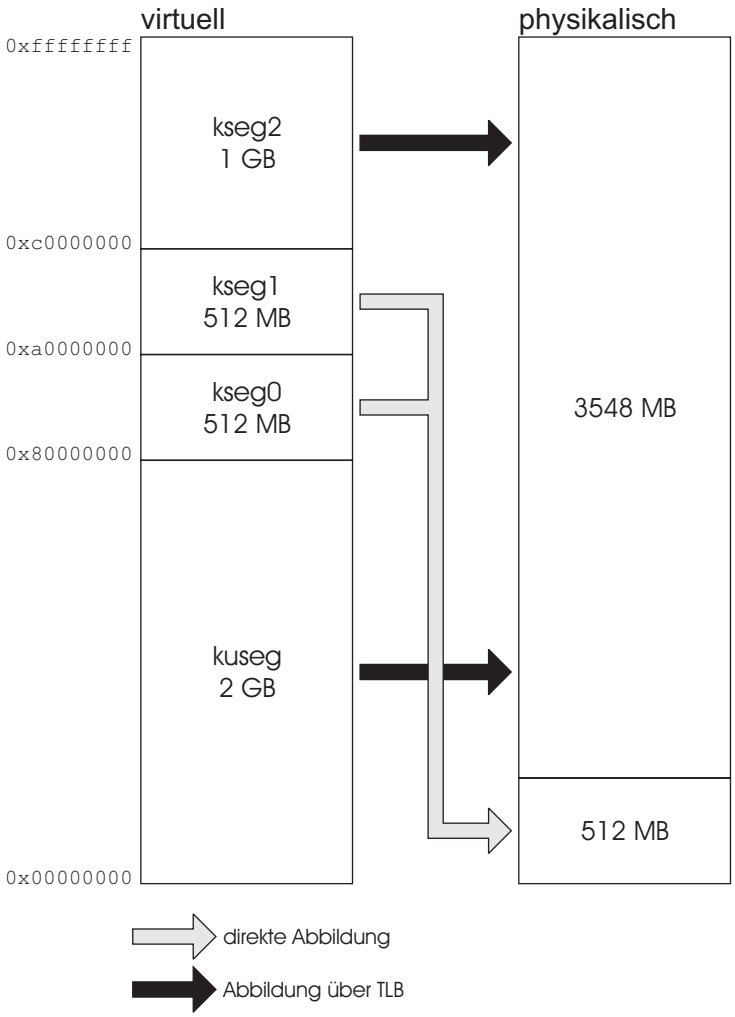

Abbildung 7.3: Mapping des MIPS R3000 Prozessors

Speichermanager-Stack und Threadmanager-Stack erstellt. Eine letzte Region beinhaltet noch den freien Speicher im Kern.

Im Benutzeradressraum werden Regionen für den Benutzer-Code, Benutzer-Daten und den freien Speicher angelegt (siehe Abbildung 7.2 auf Seite 38).

Die Funktion `vmInitRegion` dient dazu, so eine Region anzulegen.

```
typedef enum {VM_FREED, VM_ALLOCATED} RegionStatus;

typedef enum {READ_WRITE_REGION, READ_ONLY_REGION,
              PROTECTED_REGION} ProtectionMode;
```

```
static void vmInitRegion(List list, Address startAddr,
                         unsigned long int size,
                         RegionStatus status,
                         ProtectionMode pmode,
                         ThreadId owner);
```

`list:` Topsy verwaltet für jeden Adressraum eine mit freien und eine mit belegten Regionen. Je nachdem, ob es sich bei der neuen Region um eine freie oder eine belegte Region handelt, wird das hier angegeben.

`startAddr:` Anfangsadresse der Region.

`size:` Größe der Region.

`status:` Der Status gibt an, ob die Region frei oder belegt ist.

`pmode:` Modus, wie auf die Region zugegriffen werden kann. Auf eine Region kann entweder lesend und schreibend, nur lesend oder gar nicht zugegriffen werden.

`owner:` Besitzer der Region.

7.2.2 Verwaltung

Um die Verwaltung des virtuellen Speichers kümmern sich Funktionen in MMVirtualMemory.c.

Die Informationen über die verschiedenen Adressräume werden in der Variablen `addressSpaces` vom Typ `AddressSpaceDesc` gehalten. Bei Topsy gibt es zwei Adressräume – den Benutzeradressraum und den Kernadressraum.

```
static AddressSpaceDesc addressSpaces[ADDRESSSPACES];

typedef struct AddressSpaceDesc_t {
    List               regionList;
    List               freeList;
    unsigned long int  startPage;
    unsigned long int  endPage;
} AddressSpaceDesc;
```

`regionList:` Liste der belegten Regionen.

`freeList:` Liste der freien Regionen.

`startPage:` Erste virtuelle Seite des Adressraums.

`endPage:` Letzte virtuelle Seite des Adressraums.

In den beiden Listen befinden sich Einträge vom Typ `RegionDesc`.

```
typedef struct RegionDesc_t {
    unsigned long int  startPage;
    unsigned long int  numOfPages;
    ProtectionMode     pmode;
    ThreadId           owner;
} RegionDesc;
```

Die Variablen `startPage` und `numOfPages` geben den Beginn und die Länge einer Region an, `pmode` gibt an, wie auf die Region zugegriffen werden kann (lesend und schreibend, nur lesend, kein Zugriff). Die Thread-ID des Besitzers wird in der Variablen `owner` festgehalten.

7.3 Translation Lookaside Buffer

Der TLB hat die Aufgabe, zu einer virtuellen Adresse die zugehörige physikalische Adresse zu bestimmen. Abbildung 7.4 soll diese Aufgabe verdeutlichen.

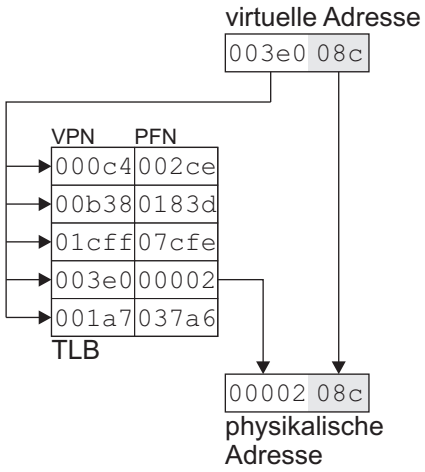

Abbildung 7.4: Aufgabe des TLB

Der TLB ist ein voll assoziativer Puffer mit einer festen Anzahl von Einträgen. Beim MIPS R3000 Prozessor bietet der TLB Platz für 64 Einträge. Es können

damit also 64·4 kB = 256 kB adressiert werden. Wenn zu einer virtuellen Adresse eine physikalische Adresse angefordert wird, es aber im TLB keinen entsprechenden Eintrag gibt, dann gibt es eine Exception und der TLB-Miss-Handler, ein Teil des Betriebssystems, muss in einer Seiten-Kachel-Tabelle die zugehörige physikalische Adresse suchen, einen Eintrag aus dem TLB entfernen (mit einer geeigneten Ersetzungsstrategie) und den neuen Eintrag in den TLB schreiben.

7.3.1 Aufbau des TLB

Abbildung 7.5 zeigt den Aufbau des TLBs und die Bedeutung der verschiedenen Register[1].

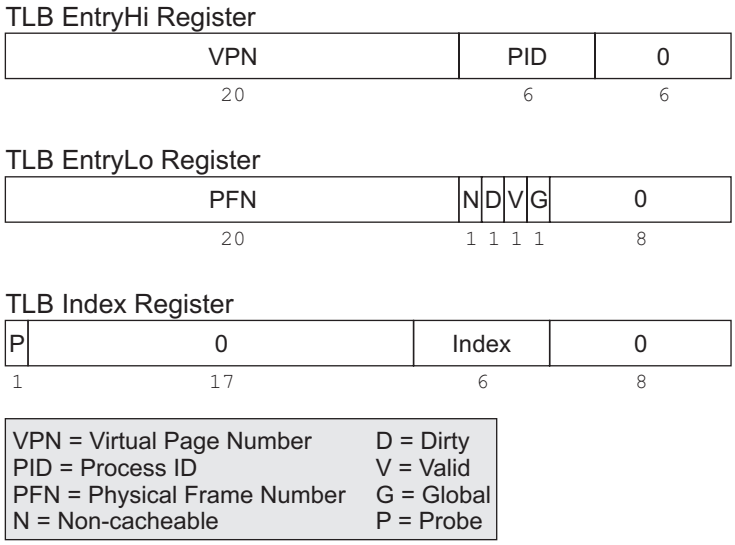

Abbildung 7.5: Aufbau des TLB beim MIPS R3000

Die oberen 20 Bits (VPN) im Register EntryHi enthalten die virtuelle Seitennummer. Die nächsten 6 Bits (PID) können die ID des Prozesses enthalten, für den diese Zuordnung gültig ist. Die unteren 6 Bits sind reserviert und müssen 0 sein.

Die oberen 20 Bits (PFN) im Register EntryLo enthalten die physikalische Seitennummer. An diese Nummer folgen 4 Bits, die gesetzt werden können, wenn die Seite nicht cacheable (N), dirty (D), valid (V) oder global (G) ist. Global bedeutet, dass die Prozess-ID nicht beachtet wird, d. h. die Beziehung zwischen virtuellen und physikalischen Seiten ist für alle Prozesse gültig. Dirty bedeutet hier beim MIPS R3000 nicht, dass die Seite geändert wurde, sondern dass die Seite verändert werden darf. Die unteren 8 Bits sind reserviert und müssen 0 sein.

[1] Diese Abbildung und alle Beispiele und Funktionen zum TLB beziehen sich auf den MIPS R3000 Prozessor

Das höchstwertige Bit (P) im Register Index ist ein Status-Bit, welches angibt, ob eine TLB-Probe-Instruktion erfolgreich verlief, ob es also einen Eintrag im TLB gibt, der mit dem Register EntryHi übereinstimmt. Die folgenden 17 Bits sowie die unteren 8 Bits sind reserviert und müssen 0 sein. Die Bits 8 bis 13 enthalten schließlich den Index.

7.3.2 Schreiben von TLB-Einträgen

Mit der Funktion `setR3kTLBEntry` kann beim MIPS R3000 Prozessor ein TLB-Eintrag gesetzt werden.

```
void setR3kTLBEntry(Register TLBEntryLow,
                    Register TLBEntryHigh,
                    Register Index);
```

Dieser Befehl erwartet als Parameter einen Wert für das Low-Register, einen Wert für das High-Register und den Index des TLB-Eintrags. Um dem Prozessor die Aufgabe zu überlassen, an welche Position der neue Eintrag geschrieben wird, kann statt dem Index einfach `TLB_RND_MASK_R3k` angegeben werden.

7.3.3 Auslesen von TLB-Einträgen

Eine Funktion zum Auslesen des TLB gibt es nicht, aber mit folgendem Eintrag in Memory/mips/tlb.S wäre das für den MIPS R3000 Prozessor möglich:

```
FRAME(getR3kTLBEntry)

sll     a2, a2, TLB_IND_INX_SHIFT_R3k /* Links-Shift */
mtc0    a2, c0_tlbind   /* Index ins Register c0_tlbind */
nop                     /* Warten */
tlbr                    /* Inhalt des TLB lesen */
nop                     /* Wieder warten */
mfc0    t0, c0_tlblo    /* Informationen auslesen */
mfc0    t1, c0_tlbhi
sw      t0, (a0)
sw      t1, (a1)

END(getR3kTLBEntry)
```

Der zugehörige Eintrag in Memory/mips/tlb.h sieht so aus:

```
void getR3kTLBEntry(Register *TLBEntryLow,
                    Register *TLBEntryHigh,
                    Register Index);
```

Wenn es einen TLB-Miss gibt, weil zu einer virtuellen Seite im TLB keine physikalische Seite gefunden wird, speichert der Prozessor die angeforderte virtuelle Seite im EntryHi Register. Um dieses Register auszulesen, kann folgende Funktion verwendet werden:

```
FRAME(getR3kTLBEntryLoHi)

mfc0    t0, c0_tlblo
mfc0    t1, c0_tlbhi
sw      t0, (a0)
sw      t1, (a1)

END(getR3kTLBEntryLoHi)
```

Und auch hier ist wieder ein Eintrag in Memory/mips/tlb.h nötig:

```
void getR3kTLBEntryLoHi(Register *TLBEntryLow,
                        Register *TLBEntryHigh);
```

7.3.4 TLB-Miss

Wenn im TLB zu einer virtuellen Seite keine passende physikalische Seite gefunden wird, dann wird eine TLB-Miss-Exception ausgelöst und der Exception-Handler `mmUTLBError` wird aufgerufen. Dieser Exception-Handler enthält jedoch noch keine Funktionalität, da Topsy den TLB nur einmal initialisiert (`mmInitMemoryMapping`) und dann nicht mehr verändert.

Um einen größeren Benutzeradressraum zu verwenden, müsste das Betriebssystem eine Seiten-Kachel-Tabelle verwalten, die zu den virtuellen Seiten die zugehörigen physikalischen Seiten verwaltet.

Bei einem TLB-Miss müsste dann das Register EntryHi ausgelesen werden. Die oberen 20 Bits dieses Registers geben die angeforderte virtuelle Seite an. Aus der Seitenkacheltabelle müsste zu der virtuellen Seite die physikalische Seite gesucht werden. Die Informationen müssen dann in den TLB eingetragen werden.

Ein solcher TLB-Miss-Handler könnte wie folgt aussehen:

```
static void mmUTLBError(ThreadId currentThread) {
  Register lo, hi;
  unsigned int virt_page; // Angeforderte virtuelle Seite
  unsigned int phys_page; // Physikalische Seite

  getR3kTLBEntryLoHi(&lo, &hi);
  virt_page = hi >> 12;

  /* search durchsucht die Seitenkacheltabelle und
   * liefert zu einer virtuellen Seite die dazugehörige
   * physikalische Seite. Falls keine solche Seite
   * gefunden wird, sollte search eine freie
   * physikalische Seite zurückliefern */
  phys_page = search(virt_page);

  lo = ((phys_page << 12) & TLB_LO_PFN_MASK_R3k) |
        TLB_VALID_R3k | TLB_GLOBAL_R3k | TLB_DIRTY_R3k;

  setR3kTLBEntry(lo, hi, TLB_RND_MASK_R3k);
}
```

7.4 Heap

Der Heap wird durch die Funktion `hmInit` initialisiert, welche von `mmInit` aufgerufen wird. Der Heap beginnt direkt nach der Kern-Daten Region (siehe Abbildung 7.2 auf Seite 38) und ist bei Topsy 64 kB groß (`KERNELHEAPSIZE`, Topsy/Configuration.h). Auf dem Heap kann der Kern Daten speichern, für Benutzerprogramme ist der Kern-Heap nicht zugänglich. Deshalb gibt es auch keine System-Calls, um Speicher auf dem Heap zu allokieren bzw. freizugeben.

Bei der Initialisierung des Heaps durch die Funktion `hmInit` werden lediglich einige Datenstrukturen initialisiert. Der Heap wird, ähnlich wie der virtuelle Speicher, mithilfe einer verketteten Liste verwaltet. Jedes Element dieser Liste enthält dabei den Status des Bereichs. Zu Beginn gibt es genau zwei Elemente. Ein Element (`start`) befindet sich am Anfang des Heaps, das zweite Element (`end`) am Ende. Die Größe eines Bereichs kann also durch die Adressen des Elements und dessen Nachfolgers berechnet werden.

```
typedef enum {HM_FREED, HM_ALLOCATED} AreaStatus;

typedef struct HmEntryDesc_t {
  AreaStatus                    status;
  struct HmEntryDesc_t*         next;
```

```
} HmEntryDesc;

typedef HmEntryDesc* HmEntry;
```

7.4.1 Allokieren von Heap-Speicher

Das Allokieren von Heap-Speicher geschieht durch die Funktion `hmAlloc`.

```
Error hmAlloc(Address* addressPtr, unsigned long int size);
```

Diese Funktion versucht, Speicher auf dem Heap zu allokieren. Dazu wird mithilfe der Funktion `findFree` die Liste durchlaufen und jedes Element wird daraufhin untersucht, ob es einen freien Speicherbereich darstellt und ob dieser Bereich genügend Platz zur Verfügung stellt. Der erste freie Bereich, der genügend Speicherplatz bietet, wird verwendet. Die Funktion `findFree` liefert bei Erfolg einen Zeiger auf ein Listenelement zurück.

Falls kein passendes Element gefunden wird, wird mit der Funktion `compact` der Speicher defragmentiert. Dabei werden mehrere aufeinanderfolgende freie Bereiche zu einem zusammengefasst. Anschließend wird noch einmal die Funktion `findFree` aufgerufen. Falls jetzt immer noch nicht genügend Platz vorhanden ist, liefert die Funktion `hmAlloc` den Wert `HM_ALLOCFAILED` zurück.

Wurde ein passendes Element gefunden, dann wird es mit der Funktion `split` in einen belegten und einen freien Teil aufgeteilt. Bei Erfolg liefert die Funktion `hmAlloc` den Wert `HM_ALLOCOK` zurück.

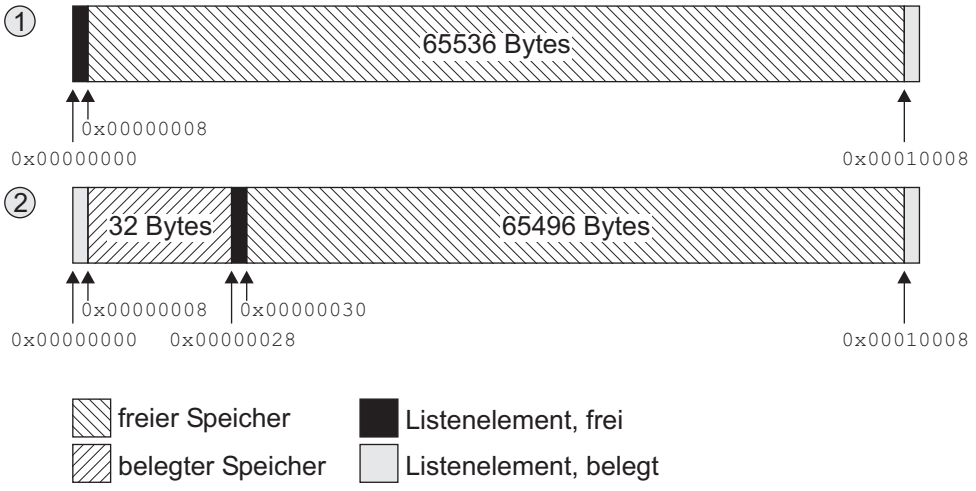

Abbildung 7.6: Heap

Abbildung 7.6 zeigt, was geschieht, wenn neuer Speicher allokiert wird. Zu Beginn (1) ist auf dem Heap noch kein Speicher allokiert und die gesamten 64 kB stehen zur Verfügung. Nachdem beispielsweise 32 Bytes allokiert wurden (2) wird der freie Bereich in einen belegten und einen freien Bereich aufgeteilt. Die Größe des allokierten Speicherbereichs lässt sich aus der Differenz der Adressen der Elemente berechnen, also `0x00000028` - `0x00000000` = 40 Bytes. Von diesen 40 Bytes müssen noch die 8 Bytes abgezogen werden, die das Listenelement belegt, also bleiben 32 Bytes übrig.

Natürlich beginnt der Heap bei Topsy nicht an Adresse `0x00000000`, sondern erst nach der Kern-Daten Region (siehe Abbildung 7.2 auf Seite 38).

7.4.2 Freigeben von Heap-Speicher

Zur Freigabe von Heap-Speicher gibt es die Funktion `hmFree`.

```
Error hmFree(Address address);
```

Diese Funktion überprüft zunächst, ob sich die übergebene Adresse im Heap-Bereich befindet. Anschließend wird die Liste durchlaufen und es wird überprüft, ob es einen Eintrag gibt, der zur übergebenen Adresse passt. Falls kein Eintrag gefunden wird, dann liefert die Funktion `HM_FREEFAILED` zurück, ansonsten wird der Eintrag als frei markiert und es wird `HM_FREEOK` zurückgeliefert.

Das Element selbst wird bei diesem Vorgang nicht gelöscht. Das geschieht erst beim Aufruf der Funktion `compact` beim Allokieren von Heap-Speicher.

7.5 System-Calls

7.5.1 Allokieren von Speicher

Der System-Call `vmAlloc` allokiert `size` Bytes Speicher im virtuellen Adressraum. Dabei wird `size` immer auf ein Vielfaches der virtuellen Seitengröße aufgerundet.

```
SyscallError vmAlloc(Address *addressPtr,
                     unsigned long int size);
```

Die Position des allokierten Speichers wird in `addressPtr` zurückgeliefert. Die Funktion `vmAlloc` liefert `VM_ALLOCOK` bei Erfolg zurück, sonst `VM_ALLOCFAILED`.

Der System-Call veranlasst das Senden einer entsprechenden Nachricht an den Speichermanager. Dieser leitet den Aufruf an `mmVmAlloc` weiter.

```
Error mmVmAlloc(Address* addressPtr, unsigned long int size,
                ThreadId owner);
```

Aus der Liste der freien Regionen wird das erste Element herausgesucht und es wird versucht, in dieser Region einen Speicher zu allokieren. Falls in der Region kein Speicher angelegt werden kann, beispielsweise, weil sie zu klein ist, wird die nächste Region aus der Liste herausgesucht. Die Funktion `mmVmAlloc` ruft zum Allokieren des Speichers die Funktion `vmAllocRegion` auf.

```
static Boolean vmAllocRegion(Region region, Page pages,
                             List list, List freeList,
                             ThreadId sender,
                             Address* addr);
```

Die Funktion `vmAllocRegion` prüft, ob es genügend freie Seiten in der Region `region` gibt. Falls genügend freie Seiten vorhanden sind, wird die freie Region in eine freie und eine belegte Region aufgeteilt. Die neu erstellte belegte Region wird in `list` eingetragen, die alte Region wird lediglich modifiziert. Falls die Liste der freien Regionen nur eine Region enthält und die gesamte Region belegt wird, wird die freie Region aus `freeList` entfernt und als belegte Region in `list` eingefügt. Die Liste `freeList` enthält dann keine Elemente mehr.

Um die Verwaltungsinformationen über die neue Region speichern zu können, muss noch Speicher auf dem Heap allokiert werden. Zu diesem Zweck wird die Funktion `hmAlloc` aufgerufen.

7.5.2 Freigeben von Speicher

Mit `vmFree` kann allokierter Speicher wieder freigegeben werden.

```
SyscallError vmFree(Address address);
```

Auch dieser System-Call wird an den Speichermanager weitergeleitet und von da aus an die Funktion `mmVmFree`.

```
Error mmVmFree(Address address, ThreadId sender);
```

Die Funktion bestimmt zunächst durch Aufruf von `getRegion` die zu der angegebenen Adresse zugehörige Region. Anschließend wird überprüft, ob der Thread den Speicher freigeben darf. Ein Benutzer-Thread darf beispielsweise nur den von

ihm allokierten Speicher freigeben. Kern-Threads dürfen auch fremden Speicher freigeben. Anhand des Speichers wird auch der zugehörige Adressraum ermittelt (Benutzer- oder Kernadressraum). Um die Region freizugeben, wird die Funktion `vmFreeRegion` aufgerufen.

```
static Boolean vmFreeRegion(List list, List freeList,
                            Region region,
                            Address address,
                            ThreadId sender);
```

Die Region wird aus der Liste der belegten Regionen gelöscht und an den Anfang der Liste der freien Regionen eingefügt. Anschließend wird noch `mmUnmapPages` aufgerufen, um eventuell gemappte Seiten zu entfernen. Diese Funktion hat allerdings unter Topsy keine Wirkung, da es kein Mapping gibt.

7.5.3 Bewegen von Speicher

Um Speicher an eine andere Adresse oder sogar in einen anderen Adressraum zu bewegen, gibt es den System-Call `vmMove`.

```
SyscallError vmMove(Address *addressPtr, ThreadId newOwner);
```

Als Parameter erwartet dieser System-Call die Adresse des Speicherbereiches und die Thread-ID des neuen Besitzers. Um den Speicher zu bewegen, wird zunächst ein neuer Speicherbereich für den Ziel-Thread allokiert, der Speicher wird kopiert, der alte Speicher wird mit Nullen überschrieben und dann freigegeben.

7.5.4 Sonstige System-Calls

Mit dem System-Call `vmCleanup` wird der gesamte Adressraum des angegebenen Threads freigegeben. Dazu wird die Liste der belegten Regionen durchlaufen und für jede Region wird `mmVmFree` aufgerufen.

```
SyscallError vmCleanup(ThreadId threadId);
```

Der Aufruf `vmProtect` sollte den Zugriffsmodus für eine Seite setzen. Der Aufruf wird an `mmVmProtect` und `mmProtectPage` weitergeleitet.

```
SyscallError vmProtect(Address startAddress,
                       unsigned long int size,
                       ProtectionMode pmode);
```

Kapitel 8

Das Modul Topsy

Das Modul Topsy enthält mehrere Hilfs-Module, die von anderen Modulen benutzt werden (z. B. verkettete Liste, Hash-Tabellen). Außerdem stellt das Modul Topsy die verschiedenen System-Calls zur Verfügung.

8.1 System-Calls

Mithilfe der System-Calls können Benutzerprogramme auf Funktionen des Kerns zugreifen. Die System-Calls sind also die einzige Verbindung zwischen den Benutzerprogrammen und den Modulen Memory, Threads und IO.

Die System-Calls werden als Nachrichten an den betreffenden Teil des Kerns geschickt. Dabei wird der System-Call meistens in eine geeignete Nachricht verpackt und mit dem Befehl `genericSyscall` weitergeleitet.

```
SyscallError genericSyscall(ThreadId to, Message* message,
                            Message* reply);
```

Die Funktion `genericSyscall` sendet eine Nachricht `message` an den Thread `to` und wartet dann auf eine Antwort, welche in `reply` gespeichert wird.

Wenn ein System-Call aufgerufen wird, dann wird dieser als Nachricht an den zuständigen Kern-Thread geschickt. Beim Versenden einer Nachricht wird eine Exception ausgelöst und der Message-Dispatcher wird aufgerufen. Dieser kopiert dann die Nachricht und fügt sie in die Warteschlange des Ziel-Threads ein. Anschließend wird der Ziel-Thread aufgeweckt, der bis dahin möglicherweise blockiert war.

In den nachfolgenden Abschnitten werden die verfügbaren System-Calls aufgelistet. Details dazu gibt es in den Kapiteln zu den jeweiligen Modulen.

8.1.1 Memory

- `SyscallError vmAlloc(Address *addressPtr, unsigned long int size);`
  Allokiert Speicher.

- `SyscallError vmFree(Address address);`
  Gibt allokierten Speicher wieder frei.

- `SyscallError vmMove(Address *addressPtr, ThreadId newOwner);`
  Bewegt Speicher in einen anderen Bereich.

- `SyscallError vmProtect(Address startAddress, unsigned long int size, ProtectionMode pmode);`
  Setzt den Modus, wie auf den Speicher zugegriffen werden kann.

- `SyscallError vmCleanup(ThreadId threadId);`
  Gibt den gesamten Speicher eines Threads frei.

8.1.2 Threads

- `SyscallError tmStart(ThreadId* id, ThreadMainFunction function, ThreadArg parameter, char *name);`
  Erzeugt einen neuen Thread.

- `SyscallError tmKill(ThreadId id);`
  Terminiert einen Thread.

- `void tmYield();`
  Mit diesem Befehl kann ein Thread seine eigene Ausführung zugunsten eines anderen Threads vorzeitig anhalten.

- `void tmExit();`
  Beendet einen Thread.

- `SyscallError tmGetInfo(ThreadId about, ThreadId* tid, ThreadId* ptid);`
  `SyscallError tmGetFirst(ThreadInfo* info);`
  `SyscallError tmGetNext(ThreadInfo* info);`
  `SyscallError tmGetThreadByName(char* name, ThreadId* tid);`
  `SyscallError tmGetIDInfo(ThreadId about, ThreadInfo* info);`
  Mit diesen Funktionen können Informationen über andere Threads ermittelt werden.

- `SyscallError tmGetTime(unsigned long* seconds,`
  `unsigned long* microSeconds);`
  Liefert die aktuelle Zeit zurück.

- `SyscallError tmSetTime(unsigned long seconds,`
  `unsigned long microSeconds);`
  Setzt die Zeit.

- `SyscallError tmMsgSend(ThreadId to, Message *msg);`
  Sendet eine Nachricht an den Thread `to`.

- `SyscallError tmMsgRecv(ThreadId* from,`
  `MessageId msgId,`
  `Message* msg,`
  `int timeOut);`
  Empfängt eine Nachricht vom Typ `msgId` vom Thread `from`. Wenn der Sender und/oder Nachrichten-Typ unwichtig sind, dann kann für den Nachrichten-Typ `ANYMSGTYPE` und für den Sender `ANY` eingesetzt werden. Die Variable `msg` zeigt nach Rückkehr auf die Nachricht. Als letzter Parameter kann ein Timeout (in Millisekunden) oder der Wert `INFINITY` eingesetzt werden, falls der Thread unendlich lange auf eine Nachricht warten soll.

8.1.3 IO

- `SyscallError ioOpen(int deviceNumber, ThreadId* id);`
  Öffnet einen Treiber.

- `SyscallError ioClose(ThreadId id);`
  Schließt einen Treiber.

- `SyscallError ioRead(ThreadId id, char* buffer,`
  `unsigned long int* nOfBytes);`
  Liest Daten von einem Treiber.

- `SyscallError ioWrite(ThreadId id, char* buffer,`
  `unsigned long int* nOfBytes);`
  Sendet Daten an den Treiber.

- `SyscallError ioInit(ThreadId id);`
  Initialisiert den Treiber.

8.2 Verkettete Listen

Die Dateien Topsy/List.c sowie Topsy/List.h stellen Funktionen zur Verfügung, um mit verketteten Listen zu arbeiten. Verkettete Listen werden beispielsweise vom Scheduler verwendet, um die einzelnen Warteschlangen zu verwalten, oder vom Speicher-Manager zur Verwaltung des virtuellen Speichers.

Um verkettete Listen zu implementieren, werden folgende Strukturen verwendet:

```
typedef struct ListElement_t {
    void* item;
    struct ListElement_t* next;
    struct ListElement_t* prev;
} ListElementDesc;

typedef ListElementDesc* ListElement;

typedef struct ListDesc_t {
    ListElement first;
    ListElement current;
    ListElement last;
} ListDesc;

typedef ListDesc* List;
```

Die Liste ist vom Typ `ListDesc`. In der Struktur `ListDesc` wird jeweils ein Zeiger auf das erste (`first`), letzte (`last`) und aktuelle (`current`) Listenelement gespeichert. Die Listenelemente selbst enthalten einen Zeiger auf die Daten (`item`), auf das vorherige (`prev`) und auf das nächste Listenelement (`next`).

Wie Abbildung 8.1 zeigt, handelt es sich bei Topsy um doppelt verkettete Listen, d. h. jedes Listenelement enthält einen Zeiger auf seinen Vorgänger und seinen Nachfolger.

8.2.1 Erzeugen einer neuen Liste

Um eine neue Liste zu erzeugen, muss die Funktion `listNew` aufgerufen werden. Diese Funktion liefert als Ergebnis einen Zeiger auf einen Listen-Deskriptor zurück. Dieser Deskriptor wird später benötigt, um die Liste zu bearbeiten.

```
List listNew();
```

Bei einem Fehler liefert die Funktion `NULL` zurück.

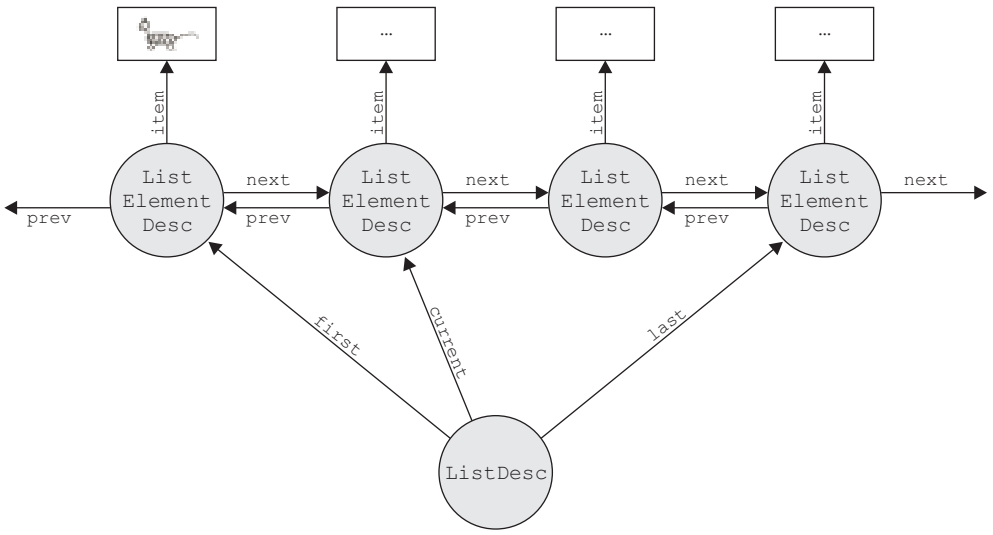

Abbildung 8.1: Doppelt verkettete Listen

8.2.2 Löschen der Liste

Die verkettete Liste kann mit dem Befehl `listFree` wieder gelöscht werden. Dabei wird die gesamte Liste durchlaufen und der Speicher, der von den Listenelementen und der Liste belegt wird, wird freigegeben. Es wird jedoch nicht der Speicher freigegeben, der von den Daten belegt wird.

```
Error listFree(List list);
```

Die Funktion liefert bei Erfolg `LIST_OK` zurück, ansonsten `LIST_ERROR`.

8.2.3 Einfügen von Elementen

Um Elemente in die Liste einzufügen, werden die Funktionen `listAddInFront` und `listAddAtEnd` zur Verfügung gestellt.

```
Error listAddInFront(List list, void* item, Address* hint);
Error listAddAtEnd(List list, void* item, Address* hint);
```

`list:` Zeiger auf ein Listen-Deskriptor Objekt. Dieser Zeiger wird von der Funktion `listNew` zurückgeliefert.

`item:` Zeiger auf die Daten.

`hint`: Zeiger auf eine Adresse, an welche die Position des Listenelements geschrieben werden kann. Die Adresse des Listenelements wird in anderen Funktion verwendet, um direkt auf das Element zuzugreifen, ohne die Liste durchsuchen zu müssen. Die einzige Funktion, welche diese Adresse zwingend benötigt, ist `listSwap`. Falls die Adresse des Listenelements für den Benutzer der Liste unwichtig ist, kann `hint` auch `NULL` sein.

Diese Funktionen liefern bei Erfolg `LIST_OK` zurück, ansonsten `LIST_ERROR`.

8.2.4 Löschen von Elementen

Mit der Funktion `listRemove` können Elemente wieder aus der Liste gelöscht werden.

```
Error listRemove(List list, void* item, Address hint);
```

Das Listenelement, welches auf die Daten `item` verweist wird aus der Liste `list` entfernt. Falls `hint` gültig ist, kann direkt auf das Element zugegriffen werden, ansonsten wird die Liste durchlaufen und das passende Element wird gesucht. Wird das Element gefunden, dann wird es aus der Liste gelöscht.

Bei einem Fehler liefert die Funktion `LIST_ERROR` zurück, ansonsten `LIST_OK`.

Der Aufruf der Funktion `listRemove` bewirkt nur das Löschen des Listenelements, die Daten selbst werden nicht gelöscht.

8.2.5 Durchsuchen der Liste

Mit den Funktionen `listGetFirst` und `listGetNext` kann die gesamte verkettete Liste durchsucht werden.

```
Error listGetFirst(List list, void** itemPtr);
Error listGetNext(List list, void** itemPtr);
```

`list`: Zeiger auf ein Listen-Deskriptor Objekt.

`itemPtr`: Dieser Zeiger wird auf das Listenelement bzw. auf die eigentlichen Daten gesetzt.

Auch diese Funktionen liefern bei Erfolg `LIST_OK`, ansonsten `LIST_ERROR` zurück.

Die aktuelle Position wird in der Liste gespeichert (`list->current`). Falls das aktuelle Element gelöscht wird, wird `list->current` auf das nächste Element gesetzt.

8.2.6 Sonstige Funktionen

Um mit verketteten Listen zu arbeiten, gibt es noch die Funktionen `listMoveToEnd` und `listSwap`.

```
Error listMoveToEnd(List list, void* item, Address hint);
Error listSwap(List listFrom, List listTo,
               void* item, Address hint);
```

Die Funktion `listMoveToEnd` bewegt ein Element ans Ende der Liste.

Die Funktion `listSwap` dient dazu, ein Element aus einer Liste `listFrom` zu entfernen und an den Anfang der Liste `listTo` einzufügen. Hier muss auch die Adresse des zu verschiebenden Elements (`hint`) übergeben werden, da die Funktion `listSwap` direkt auf das Element zugreift und nicht die Liste durchsucht.

Diese Funktionen liefern bei Erfolg `LIST_OK` zurück, ansonsten `LIST_ERROR`.

8.3 Hash-Tabellen

Hash-Listen bzw. Hash-Tabellen dienen dazu, bestimmte Elemente mittels Schlüssel zu verwalten und schnell zu finden. Der Vorteil gegenüber einer verketteten Liste liegt darin, dass nicht die ganze Liste durchsucht werden muss, um ein Element zu finden. Im Idealfall kann auf das Element mithilfe seines Schlüssels direkt zugegriffen werden.

Hash-Tabellen werden mit folgenden Strukturen verwaltet:

```
typedef struct HashListElementDesc_t {
        void* item;
        unsigned long int key;
        struct HashListElementDesc_t* next;
} HashListElementDesc;

typedef HashListElementDesc* HashListElement;

typedef struct HashList_t {
        HashListElement table[HASHLISTSIZE];
} HashListDesc;

typedef HashListDesc* HashList;
```

Die Hash-Tabelle ist vom Typ `HashListDesc`. Die Liste enthält eine Tabelle mit `HASHLISTSIZE` Zeigern, wobei jeder von diesen Zeigern auf ein Element vom Typ `HashListElementDesc` zeigt. Abbildung 8.2 soll den Aufbau einer solchen Hash-Liste verdeutlichen.

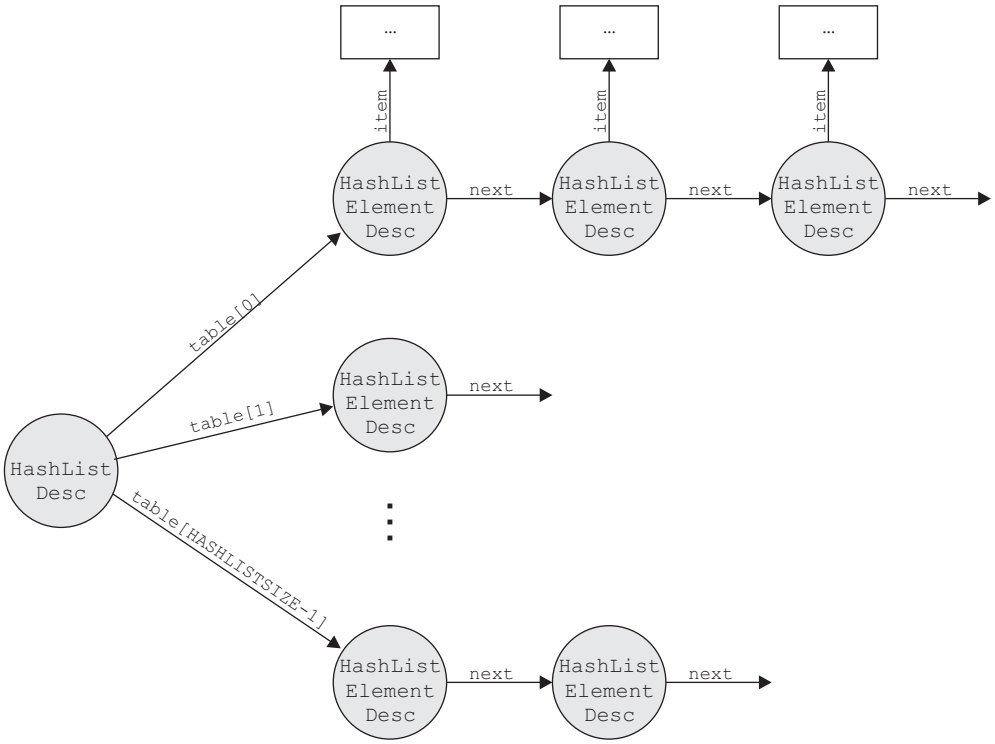

Abbildung 8.2: Hash-Tabellen

8.3.1 Erzeugen einer neuen Hash-Tabelle

Mit der Funktion `hashListNew` kann eine neue Hash-Tabelle erzeugt werden. Die Funktion liefert bei Erfolg einen Zeiger auf die neue Liste zurück, ansonsten `NULL`.

```
HashList hashListNew();
```

Beim Aufruf dieser Funktion wird die neue Tabelle initialisiert und alle Zeiger (`table[i]`) werden zunächst auf `NULL` gesetzt, da noch keine Elemente in der Tabelle vorhanden sind.

8.3.2 Löschen der Hash-Tabelle

Der von der Hash-Tabelle belegte Speicher kann mit der Funktion `hashListFree` freigegeben werden. Dabei wird, wie bei den verketteten Listen, nicht der Spei-

cher freigegeben, der von den Daten belegt wird, sondern nur der Heap-Speicher, der beim Erzeugen der Hash-Tabelle und deren Verwaltungsstrukturen belegt wurde.

```
Error hashListFree(HashList list);
```

Allerdings wird diese Funktion, ebenso wie die Funktion zum Löschen einer verketteten Liste, in keinem der Headerfiles exportiert und keine dieser Funktionen wird jemals aufgerufen.

8.3.3 Einfügen von Elementen

Mit der Funktion `hashListAdd` können neue Elemente in die Liste eingefügt werden.

```
Error hashListAdd(HashList list, void* data,
                  unsigned long int key);
```

Die Funktion fügt die Daten `data` über den Schlüssel `key` in die Liste `list` ein.

Das neue Element muss in die Tabelle eingefügt werden. Dazu muss jedoch zuvor die Position berechnet werden, an welche das Element eingefügt wird. Dies geschieht mithilfe einer geeigneten Hash-Funktion. Falls die Position in der Tabelle bereits belegt ist, werden die Elemente dort als einfach verkettete Liste gespeichert. Das neue Element wird dabei ans Ende der Liste eingefügt. Gleichzeitig wird überprüft, ob sich bereits ein Element mit demselben Schlüssel in der Tabelle befindet. Falls sich bereits ein Element mit demselben Schlüssel in der Hash-Tabelle befindet, wird das neue Element nicht eingefügt und die Funktion liefert `HASHDUPLICATEKEY` zurück. Bei anderen Fehlern wird `HASHERROR` zurückgeliefert, bei Erfolg `HASHOK`.

8.3.4 Löschen von Elementen

Elemente können aus der Hash-Tabelle mit der Funktion `hashListRemove` entfernt werden. Dazu muss nur der Schlüssel bekannt sein.

```
Error hashListRemove(HashList list, unsigned long int key);
```

Die Funktion durchsucht die Liste an der errechneten Position in der Tabelle und löscht das Element, falls es gefunden wird.

Die Funktion liefert `HASHNOTFOUND` zurück, falls das Element nicht gefunden wird, sonst `HASHOK`.

8.3.5 Suchen von Elementen

Nach Elementen kann mit der Funktion `hashListGet` gesucht werden.

```
Error hashListGet(HashList list, void** data,
                  unsigned long int key);
```

Als Parameter benötigt diese Funktion die Hashliste (`list`) und einen Schlüssel (`key`). Wenn das gesuchte Element gefunden wird, dann zeigt `*data` auf die Daten und die Funktion liefert `HASHOK` zurück. Wird das gesuchte Element nicht gefunden, dann liefert die Funktion `HASHNOTFOUND` zurück und `*data` wird auf `NULL` gesetzt.

8.4 Spinlocks

Mit Spinlocks kann der gegenseitige Ausschluss von nebenläufigen Prozessen gewährleistet werden. Nebenläufige Prozesse sollten immer dann unter gegenseitigem Ausschluss arbeiten, wenn sie gleichzeitig auf dasselbe Objekt zugreifen und dieses Objekt möglicherweise verändern.

Der Vorteil von Spinlocks liegt darin, dass sie einfach implementiert werden können, der Nachteil ist, dass aktiv gewartet wird.

Um Spinlocks zu implementieren, wird eine atomare Test & Set Operation benötigt. Das folgende Beispiel zeigt, wie diese Test & Set Operation für den MIPS Prozessor implementiert ist:

```
FRAME(testAndSet)

.set noreorder

mfc0    t0, c0_status
mtc0    zero, c0_status
nop                     /* interrupts disabled */

lw      t1, 0(a0)       /* load lockVariable contents */
move    v0, zero        /* prepare return FALSE */
bnez    t1, lockIsAlreadySet
nop
addiu   v0, 1           /* return TRUE (successful locking) */
sw      v0, 0(a0)       /* write back locked value */
lockIsAlreadySet:
```

```
mtc0    t0, c0_status   /* re-enable interrupts */

.set reorder

END(testAndSet)
```

Etwas umgeschrieben sieht diese Funktion so aus:

```
int test_and_set (int *a0) {

/* Hier müssten zuerst Interrupts ausgeschaltet werden */
  int t1 = *a0, v0 = 0;
  if (t1 == 0) {
    v0++;
    *a0 = v0;
  }
  /* Und Interrupts wieder einschalten */
  return v0;
}
```

Dadurch, dass zu Beginn der Funktion die Interrupts gesperrt werden, wird auch der Scheduler nicht aufgerufen, der die Ausführung dieser Funktion unterbrechen könnte. Somit ist diese Funktion atomar.

Genau genommen handelt es sich bei dieser Funktion um eine Test & Test & Set Operation, denn `v0` wird erst gesetzt, wenn der vorherige Test (`t1 == 0`) erfolgreich war.

Diese Test & Set Operation wird solange durchgeführt, bis sie den Wert 1 liefert, dann erst kann der kritische Abschnitt betreten werden. Beim Verlassen des kritischen Abschnitts muss lediglich die Spinlock-Variable wieder auf `FALSE` gesetzt werden.

Zum Arbeiten mit Spinlock-Variablen werden die Funktionen `lockInit`, `lock`, `unlock` und `lockTry` zur Verfügung gestellt.

```
void lockInit(Lock lock);
void lock(Lock lock);
void unlock(Lock lock);
Boolean lockTry(Lock lock);
```

Die Spinlock-Variable selbst wird durch folgende Struktur implementiert:

```
typedef struct LockDesc_t {
    Boolean lockVariable;
} LockDesc;

typedef LockDesc* Lock;
```

Zum Initialisieren der Variablen dient die Funktion `lockInit`. Sie setzt `lockVariable` auf `FALSE`.

Vor dem Betreten eines kritischen Abschnitts muss die Funktion `lock` aufgerufen werden. Sie führt solange die Test & Set Operation durch, bis der kritische Abschnitt betreten werden kann.

Ähnlich wie die Funktion `lock` arbeitet die Funktion `lockTry`. Diese Funktion versucht genau ein Mal, ob der kritische Abschnitt betreten werden kann und kehrt dann zurück. Diese Funktion liefert `TRUE` zurück, falls der kritische Abschnitt betreten werden kann, sonst `FALSE`.

Mit der Funktion `unlock` kann man die Spinlock-Variable wieder freigeben. Diese Funktion muss beim Verlassen des kritischen Abschnitts aufgerufen werden.

8.5 Support-Funktionen

Die Datei Topsy/Support.c enthält einige Hilfsroutinen, die von anderen Kern-Threads verwendet werden können.

- `void byteCopy(Address targetAddress, Address sourceAddress, unsigned long int nbBytes);`
  Kopiert `nbBytes` von `sourceAddress` nach `targetAddress`. Falls sich die beiden Adressbereiche überlappen, kann es zu Problemen kommen. Falls der Quell-Bereich vor dem Ziel-Bereich liegt und dabei in den Zielbereich hineinragt, wird beim Kopieren ein Teil des Quell-Bereichs überschrieben und diese falschen Daten werden dann später auch in den Ziel-Bereich kopiert.

- `void longCopy(Address targetAddress, Address sourceAddress, unsigned long int nbBytes);`
  Kopiert `nbBytes` von `sourceAddress` nach `targetAddress`, wobei `nbBytes` ein Vielfaches von 4 sein muss. Auch hier dürfen sich die Adressbereiche nicht überlappen.

- `void zeroOut(Address target, unsigned long int size);`
  Füllt einen Speicherbereich der Grösse `size` mit Nullen.

- `void stringCopy(char* target, char* source );`
  Kopiert einen null-terminierten String von `source` nach `target`.

- `void stringNCopy(char* target, char* source,`
  `unsigned long int size);`
  Kopiert einen null-terminierten String von `source` nach `target`, wobei maximal `size` Zeichen kopiert werden.

- `Boolean isStringEqual(char* s1, char* s2, int maxLength);`
  Vergleicht die ersten `maxLength` Zeichen zweier Strings und gibt bei Gleichheit `TRUE` zurück, sonst `FALSE`.

- `Boolean testAndSet(Boolean* lockvar);`
  Führt eine atomare Test & Set Operation durch und gibt bei erfolgreichem Set `TRUE` zurück, sonst `FALSE`.

8.6 Fehlerbehandlung

Die Dateien Topsy/Error.c und Topsy/Error.h stellen Routinen und Makros zur Fehlerbehandlung zur Verfügung. Benutzerprogramme und auch Teile des Kerns können diese Routinen benutzen, um Fehlermeldungen, Warnungen oder einfache Informationen auf der Konsole auszugeben.

```
void panic(char* threadName, const char* fileName,
           int line, char* errorMessage);

void error(char* threadName, const char* fileName,
           int line, char* errorMessage);

void warning(char* threadName, const char* fileName,
             int line, char* errorMessage);

void info(char* threadName, const char* fileName,
          int line, char* errorMessage);

void printRegisters();
```

Um die Funktionen `panic`, `error`, `warning` und `info` bequemer nutzen zu können, sind in Error.h auch einige Makros definiert.

```
#define INFO_LEVEL      1
#define WARNING_LEVEL   2
```

```
#define ERROR_LEVEL     3

#define PANIC(s)   panic( getCurrentThreadName(), \
                          __FILE__, __LINE__, s)

#if DEBUG_LEVEL <= INFO_LEVEL
#define INFO(s)    info(getCurrentThreadName(), \
                        __FILE__, __LINE__, s)
#else
#define INFO(s)
#endif

#if DEBUG_LEVEL <= WARNING_LEVEL
#define WARNING(s) warning(getCurrentThreadName(), \
                           __FILE__, __LINE__, s)
#else
#define WARNING(s)
#endif

#if DEBUG_LEVEL <= ERROR_LEVEL
#define ERROR(s)   error(getCurrentThreadName(), \
                         __FILE__, __LINE__, s)
#else
#define ERROR(s)
#endif
```

Je kleiner der Debug-Level ist, desto mehr Meldungen werden also ausgegeben. Panic-Meldungen werden immer ausgegeben. Etwas schwächer sind Error-Meldungen gefolgt von Warning- und Info-Meldungen. Der Debug-Level wird in den architekturabhängigen Makefiles (z. B. in Makefiles/Makefile.IDT.mips) angegeben.

8.7 Konfiguration

Das Modul Topsy enthält auch Dateien, die Informationen über die verwendete Architektur enthalten. Dies sind für den MIPS-Prozessor beispielsweise die Dateien Topsy/mips/cpu.h (Informationen über Segmentierung des Speichers, TLB, ...), Topsy/mips/limits.h (Größe und Wertebereiche verschiedener Integer-Typen) und Topsy/mips/asm.h (Namen der Register).

In der architekturunabhängigen Datei Topsy/Configuration.h können einige Parameter für Topsy eingestellt werden, beispielsweise die Größe des Kern-Heaps,

Anzahl der Prioritäten für Threads, oder die Zeitdauer, die ein Thread beim preemptiven Multitasking abgearbeitet wird.

Kapitel 9

Das Modul User

Das Modul User enthält bereits einige Programme (z. B. typisches "Hello World"-Beispiel) und ist dazu da, weitere Benutzerprogramme aufzunehmen. All diese Threads laufen im Benutzeradressraum und können über System-Calls auf Funktionen des Betriebssystems zugreifen. Außerdem enthält dieses Modul einige Hilfsfunktionen, die in User/UserSupport.h definiert sind.

Da Topsy kein Dateisystem besitzt, müssen die Benutzerprogramme zusammen mit dem Kern compiliert werden.

9.1 Shell

Die Shell ist der einzige Prozess, der im Benutzeradressraum gestartet wird. Alle anderen Funktionen, die der Benutzer über die Shell startet, werden als Threads im Adressraum der Shell ausgeführt.

Die Shell ist sehr einfach aufgebaut und kennt folgende Befehle:

- `exit`: Verlassen der Shell.
- `help`, `info`: Ausgabe einer kurzen Hilfe und Auflistung der verfügbaren Benutzerprogramme.
- `ps`: Auflistung der laufenden Threads sowie deren ID und Status.
- `start <funktion> <arg>`: Starten einer Funktion mit den angegebenen Argumenten.
- `kill <id>`: Terminieren eines Threads mit der entsprechenden Thread-ID.

Die Shell bietet auch die Möglichkeit, Threads beim Start in den Hintergrund zu stellen. Dies geschieht, wie bei UNIX-Shells, durch das Symbol `&`, also beispielsweise `start hello &`.

Die Shell wird nach dem Booten vom Threadmanager gestartet. Dies geschieht durch den Sprung an eine bestimmte Adresse (0x00001A98). Nach dem Starten der Shell läuft diese in eine Endlosschleife und überprüft und bearbeitet dort die Befehle des Benutzers.

9.2 UserSupport

Das Modul UserSupport enthält einige Hilfsfunktionen, welche die Benutzerprogramme verwenden können.

- `void byteCopy(Address targetAddress, Address sourceAddress, unsigned long int nbBytes);`
  Kopiert `nbBytes` von `sourceAddress` nach `targetAddress`. Bei der Überlappung der beiden Speicherbereiche kann es zu Problemen kommen.
- `void zeroOut(Address target, unsigned long int size);`
  Füllt einen Speicherbereich mit Nullen.
- `void initmem(Address target, unsigned long int size, char c);`
  Füllt einen Speicherbereich mit einem Zeichen `c`.
- `void stringCopy(char* target, char* source);`
  Kopiert einen String von `source` nach `target`. Die beiden Speicherbereiche sollten sich nicht überschneiden.
- `void stringNCopy(char* target, char* source, unsigned long int size);`
  Kopiert maximal `size` Zeichen von einem String `source` nach `target`.
- `int stringLength(char* s);`
  Gibt die Länge eines Strings zurück.
- `void stringConcat(char *dest, char *source1, char *source2);`
  Konkateniert zwei Strings.
- `void int2string(char *str, int i);`
  Wandelt eine Zahl in einen String um.
- `void display( ThreadId tty, char* s);`
  Gibt einen String auf dem Gerät `tty` aus.

- `int stringCompare(char* a, char* b);`
  Vergleicht zwei Strings.
- `int stringNCompare(const char *s1, const char *s2, int n);`
  Vergleicht die ersten `n` Zeichen zweier Strings.
- `int power(int base, int n);`
  Liefert das Ergebnis base^n zurück.
- `int atoi(int* intValue, char* string);`
  Konvertiert einen String in eine Zahl.
- `void reverse(char s[]);`
  Dreht einen String um.
- `void itoa(int n, char s[]);`
  Konvertiert eine Zahl in einen String. Im Gegensatz zu `int2string` verwendet `itoa` die Funktion `reverse`.

9.3 Hinzufügen eigener Programme

9.3.1 Anpassen der Shell

Zuerst muss die Shell so angepasst werden, dass sie mit einem bestimmten Kommando den neuen Thread starten kann. Dazu muss in der Datei User/shell.c die Struktur `userCommands` sowie `NUSERCOMMANDS`, die Anzahl der Benutzerkommandos und Einträge dieser Struktur, angepasst werden. Die Variable `userCommands` enthält Einträge vom Typ `ShellFunction`.

```
typedef struct ShellFunction_t {
    char* name;
    ThreadMainFunction function;
    ThreadArg arg;
} ShellFunction;

typedef void* ThreadArg;
typedef void(*ThreadMainFunction)(ThreadArg);
```

Eine Funktion besitzt einen Namen `name`, eine Funktion `function` und beim Aufruf dieser Funktion werden die Parameter `arg` übergeben.

Das folgende Beispiel zeigt, wie die Funktion `neue_funktion` mit dem Namen `neu` hinzugefügt wird.

```
ShellFunction userCommands[NUSERCOMMANDS] = {
        {"neu", neue_funktion, (ThreadArg)0},
        {"shell", main, (ThreadArg)0},
        {"trap", trap, (ThreadArg)0},
        ...
};
```

Es gibt mehrere Möglichkeiten, wie einer Funktion Parameter übergeben werden können. Eine Angabe von `(ThreadArg)0` übergibt keine Parameter, eine Angabe von `(ThreadArg)"a b c"` übergibt die Parameter `"a"`, `"b"` und `"c"` und `(ThreadArg)argArray` übergibt der Funktion die Parameter, welche der Benutzer beim Starten der Funktion in der Shell eingibt. Einer Funktion können maximal `MAXNBOFARGUMENTS` Argumente übergeben werden.

Von der Shell aus kann die neue Funktion mit dem Befehl `start` ausgeführt werden. Dieser Befehl erwartet als Parameter den Namen der Funktion.

Damit die neue Funktion auch gefunden wird, muss auch die entsprechende Datei mit einer `#include`-Anweisung in User/Shell.c eingebunden werden.

9.3.2 Anpassen des Makefiles

Damit die neue Funktion auch compiliert wird, muss noch das Makefile für die Benutzerprogramme geändert werden. In der Datei Makefiles/Makefile.user muss also der Wert `USERFILES` entsprechend geändert werden.

9.3.3 Programmieren der Funktion

Der letzte Schritt besteht in der Programmierung der neuen Funktion. Die Funktion könnte wie folgt aussehen:

```
void neue_funktion ();
void neue_funktion (char* argArray[]);
```

Die Variable `argArray` enthält die übergebenen Parameter, wobei `argArray[0]` den Namen der Funktion selbst enthält. Das letzte Argument ist `NULL`.

Kapitel 10

Zusammenfassung

Topsy ist ein 32 Bit Betriebssystem mit MMU-Unterstützung, welches auf dem MIPS R3000 Prozessor läuft. Durch den einfachen, modularen Aufbau und der klaren Trennung zwischen hardwareabhängigem und hardwareunabhängigem Quellcode ist Topsy sehr gut zu Lehrzwecken geeignet.

Zusätzlich zur Dokumentation von Topsy wurde im Rahmen dieser Studienarbeit auch die Funktionalität etwas erweitert. Da der Benutzeradressraum in Topsy durch die Größe des TLB auf 256 kB begrenzt ist, wurde eine Seiten-Kachel-Tabelle eingefügt, welche die Zuordnungen von virtuellen zu physikalischen Seiten speichert. Außerdem wurde ein TLB-Miss-Handler programmiert, der immer dann aufgerufen wird, wenn die virtuelle Seite nicht im TLB gefunden wird. Der TLB-Miss-Handler durchsucht dann die Seiten-Kachel-Tabelle und fügt die benötigten Daten in den TLB ein.

Da es bei Topsy bisher nur die Möglichkeit gab, durch Spin-Locks den gegenseitigen Ausschluss zu gewährleisten, wurde das Betriebssystem noch um Semaphoren erweitert. Mit speziellen System-Calls können Semaphoren erstellt, P- und V-Operationen durchgeführt und Semaphoren gelöscht werden. Im Rahmen dieser Aufgabe wurde auch noch eine Priority Inheritance implementiert. Wenn ein höherpriorer Prozess also auf eine Semaphore wartet, die von einem niederprioren Prozess gehalten wird, dann wird dessen Priorität auf jene des höherprioren gesetzt. Bei der Freigabe der Semaphore wird die ursprüngliche Priorität wiederhergestellt.

Allerdings gab es bei Topsy nur drei verschiedene Prioritäten - eine für Kern-Threads, eine für Benutzer-Threads und eine für den Idle-Thread. Um die Funktionalität der Prioritätsvererbung zu testen, wurden zwei zusätzliche Prioritäten für Benutzer-Threads eingefügt. Mit System-Calls kann die Priorität eines Threads ermittelt und gesetzt werden.

Eine letzte große Änderung betraf den MIPS-Simulator. Dieser wurde so erweitert, dass er ein 200×200 Pixel großes Display darstellen kann. Dieses Display

kann von Topsy aus mit einem eigens dafür geschriebenen Treiber angesteuert werden.

Weitere interessante Erweiterungen wären beispielsweise die Implementierung eines Dateisystems und eine Netzwerk-Unterstützung.

Anhang A

Installation von Topsy

A.1 Benötigte Programme

Um Topsy zu installieren und damit zu arbeiten, werden folgende Programme benötigt:

- Quellcode von Topsy und des MIPS-Simulators[1]
- Quellcode des GNU C Compilers[2] , der GNU binutils[3] und des GNU Debuggers[4]
- Java-Compiler

A.2 Cross Compiler

Zunächst muss ein Compiler gebaut werden, der Code für den MIPS IDT R3000 Prozessor erzeugt, da Topsy auf diesem Prozessor läuft. Mit diesem Compiler kann dann der Topsy-Kern erzeugt werden.

A.2.1 GNU binutils

Der GNU C Compiler kann erst gebaut werden, wenn die GNU binutils für den Zielprozessor existieren. Nach dem Entpacken der binutils müssen mit folgenden Befehlen passende Makefiles generiert und die GNU binutils compiliert werden:

[1] Topsy 2.0
[2] GNU C Compiler 2.9.5
[3] GNU binutils 2.9.1
[4] GNU Debugger 4.18

```
./configure --target=mips-idt-ecoff --prefix=/usr/local[5]
make all install
```

Nach dem erfolgreichen Compilieren der GNU binutils sollte der Pfad auf das Verzeichnis /usr/local/bin gesetzt werden, damit beim Compilieren des GNU C Compilers die erzeugten Tools für den MIPS-Prozessor gefunden werden:

```
setenv PATH /usr/local/bin:$PATH
```

A.2.2 GNU C Compiler

Nach dem Erzeugen der GNU binutils kann der GNU C Compiler erzeugt werden. Vorher müssen an einigen Dateien jedoch einige Änderungen vorgenommen werden.

In der Datei configure.in sollten einige Angaben wie folgt geändert werden:

```
host_tools="gcc gas"
target_libs=""
target_tools=""
native_only=""
cross_only=""
```

Durch diese Angaben wird der Compiler davon abgehalten, unnötige Bibliotheken und Tools zu compilieren.

In den Dateien ./gcc/libgcc2.c sowie ./gcc/frame.c werden durch Include-Anweisungen die Dateien sdtlib.h und unistd.h eingebunden. Diese müssen gelöscht werden, zum Einen, weil sie nicht benötigt werden, zum Anderen, weil sie vom Compiler nicht gefunden werden.

Mit den folgenden Befehlen wird schließlich der Cross-Compiler erzeugt:

```
./configure --target=mips-idt-ecoff --prefix=/usr/local
            --with-gnu-as --with-gnu-ld
make LANGUAGES="c" all install
```

Falls `make` mit dem Fehler abbricht, dass zum Beispiel die Datei .../sys-includes/*.h nicht gefunden wird, dann genügt es, einfach eine entsprechende leere Datei zu erstellen und `make` erneut mit den obigen Optionen zu starten.

Bei Erfolg wird der Cross-Compiler in das Verzeichnis /usr/local/bin installiert.

[5] Es wird angenommen, dass das Zielverzeichnis /usr/local ist

A.2.3 GNU Debugger

Das Compilieren des GNU Debuggers gestaltet sich wieder etwas einfacher. Mit folgenden Anweisungen wird der Debugger für den MIPS-Prozessor erzeugt.

```
./configure --target=mips-idt-ecoff --prefix=/usr/local
make all install
```

Eventuell muss statt dem `make` ein `gmake` verwendet werden (unter Solaris beispielsweise). Der GNU Debugger wird bei erfolgreichem Durchlauf des Compilers in das Verzeichnis /usr/local/bin installiert.

A.3 MIPS-Simulator

Da der MIPS-Simulator in Java geschrieben wurde, ist ein Java-Compiler nötig um diesen zu compilieren. Eine Eingabe von `make` genügt, um den Simulator zu erzeugen. Anschließend kann der Simulator mit dem Befehl `java Simulator` gestartet werden. Natürlich wird auch noch ein funktionierender Topsy-Kern (topsy.srec) benötigt.

A.4 Topsy

Um Topsy zu compilieren genügt wiederum das Eintippen von `make` und das Drücken von `Return`. Falls alles glatt läuft, der Cross-Compiler richtig installiert und gefunden wird und sich keine Fehler in den Topsy-Sourcen befinden, dann sollte das Ergebnis ein funktionierender Topsy-Kern sein (topsy.srec), der darauf wartet, getestet zu werden.

Anhang B

Erweiterung von Topsy

B.1 Prioritäten

Bei Topsy gibt es nur drei verschiedene Prioritäten - eine für Kern-Thread, eine für Benutzer-Threads und eine für den Idle-Thread. Topsy wurde deshalb um zwei Prioritäten für die Benutzer-Threads erweitert.

Dazu wurde zunächst die Anzahl der Prioritäten um zwei erhöht (`NBPRIORITYLEVELS`, Topsy/Configuration.h). In der Datei Threads/tm-include.h wurden dann die beiden neuen Prioritäten (`USER_PRIORITY_HIGH` und `USER_PRIORITY_LOW`) eingefügt. Damit existierten zwar insgesamt fünf Prioritäten, aber es gab noch keine Möglichkeit, die Priorität eines Threads zu ermitteln oder zu setzen. Diese Funktionalität wurde durch das Hinzufügen zweier System-Calls erreicht.

```
SyscallError tmSetPriority ( ThreadId id, int priority);
SyscallError tmGetPriority (ThreadId id, int *priority);
```

Um System-Calls zu implementieren, müssen auch geeignete Nachrichten-Typen und -Strukturen implementiert werden. Bei den Nachrichten-Typen wurden in Topsy/Messages.h die Typen `TM_SETPRIORITY`, `TM_GETPRIORITY`, `TM_SETPRIORITYREPLY` und `TM_GETPRIORITYREPLY` hinzugefügt. Zu den möglichen Fehlermeldungen wurden die Werte `TM_SETPRIORITYOK`, `TM_SETPRIORITYFAILED`, `TM_GETPRIORITYOK` und `TM_GETPRIORITYFAILED` hinzugefügt. Zuletzt wurde noch eine Nachrichten-Struktur hinzugefügt.

```
typedef struct TMPriorityMsg_t {
  int priority;
  ThreadId id;
  MessageError errorCode;
} TMPriorityMsg;
```

B.1.1 Priorität ermitteln

Mit dem System-Call `tmGetPriority` kann die Priorität eines Threads ermittelt werden. Dazu wird eine Nachricht vom Typ `TM_GETPRIORITY` an den Threadmanager (`tmMain`) geschickt. Dieser ruft die Funktion `getPriority` in der Datei Threads/TMScheduler.c auf. Diese Funktion sucht in der Thread-Hashtabelle `threadHashList` nach dem Thread und liefert dessen Priorität zurück.

Bei Erfolg liefert die Funktion `TM_GETPRIORITYOK` zurück, bei einem Fehler `TM_GETPRIORITYFAILED`.

B.1.2 Priorität setzen

Die Priorität eines Threads kann mit dem System-Call `tmSetPriority` gesetzt werden. Auch hier wird eine Nachricht vom Typ `TM_SETPRIORITY` an den Threadmanager verschickt, der dann die Funktion `setPriority` aufruft. Diese Funktion sucht dann wieder nach dem Thread. In die Thread-Struktur wird dann die neue Priorität eingetragen. Da der Scheduler jedoch für jede Priorität eine eigene Warteschlange verwaltet, muss der Thread noch in die richtige Warteschlange eingefügt werden. Dabei muss darauf geachtet werden, ob sich der Thread momentan in der Warteschlange der blockierten oder der bereiten Threads befindet.

Bei Erfolg liefert die Funktion `TM_SETPRIORITYOK` zurück, bei einem Fehler `TM_SETPRIORITYFAILED`.

B.2 Semaphoren

Da die Verwendung von Spinlocks nur für Kern-Threads vorgesehen ist und Spinlocks durch das aktive Warten sehr rechenintensiv sind, wurden Semaphoren eingefügt. In der Datei Topsy/Semaphore.c wurden dabei folgende Funktionen implementiert:

```
void semaphoreNew(ThreadId id);
void semaphoreP(ThreadId id, unsigned int number);
void semaphoreV(ThreadId id, unsigned int number);
void semaphoreMain();
```

Die Semaphoren werden mit folgender Struktur verwaltet:

```
typedef struct SemaphoreDesc_t {
  ThreadId currentThread;
  int originalThreadPriority;
  int maxWaitingPriority;
  int number;
  int locked;
  List waitingThreads;
  Lock lock;
} SemaphoreDesc;
```

`currentThread`: Thread, der die Semaphore belegt hat.

`originalThreadPriority`: Ursprüngliche Priorität des Threads.

`maxWaitingPriority`: Priorität des höchstprioren Threads, der auf diese Semaphore wartet.

`number`: Nummer der Semaphore.

`locked`: Angabe, ob Semaphore gerade benutzt wird.

`waitingThreads`: Liste der Threads, die die P-Operation aufgerufen haben und auf die Semaphore waren.

`lock`: Lockvariable, damit die Operationen auf den Semaphoren-Strukturen und Listen unter gegenseitigem Ausschluss verlaufen.

Damit schnell auf die Informationen der einzelnen Semaphoren zugegriffen werden kann, werden diese in einer Hash-Tabelle verwaltet.

B.2.1 Erzeugen einer neuen Semaphore

Das Erzeugen einer Semaphore geschieht mit folgendem System-Call:

```
SyscallError semNew(unsigned int *number);
```

Durch Aufruf dieses System-Calls wird eine Nachricht vom Typ `SEM_NEW` an den Semaphoren-Manager `semaphoreMain` geschickt. Dieser ruft dann die Funktion `semaphoreNew` auf. Diese Funktion allokiert Speicher zur Verwaltung einer neuen Semaphoren-Struktur, erzeugt dann eine neue Semaphoren-Nummer und eine neue Liste für die wartenden Threads. Zuletzt wird die erzeugte Struktur in die Hashtabelle `semaphores` eingetragen.

Bei Erfolg liefert die Funktion den Wert `SEM_OK` zurück, ansonsten `SEM_ERROR`.

B.2.2 P-Operation

Durch folgenden System-Call signalisiert ein Thread dem System, dass er einen kritischen Abschnitt betreten möchte und dazu eine Semaphore sperren will.

```
SyscallError semP(unsigned int number);
```

Diesem System-Call muss eine gültige Semaphoren-Nummer übergeben werden. Der System-Call wird als Nachricht vom Typ `SEM_P` an den Semaphoren-Manager geschickt, der dann die Funktion `semaphoreP` aufruft. Diese Funktion sucht über die Semaphoren-Nummer die Semaphoren-Struktur in der Hashtabelle. Falls die Semaphore bereits gesperrt ist, wird der Thread in die Warteschlange eingetragen. In dieser Warteschlange werden keine Prioritäten berücksichtigt.

Falls mehrere Threads unterschiedlicher Priorität eine gemeinsame Semaphore benutzen, kann es zu einer Priority Inversion kommen. Wie in Abbildung B.1 dargestellt wird, sperrt Thread 3 zum Zeitpunkt a die Semaphore und berechnet dann etwas. Zum Zeitpunkt b kommt Thread 2 hinzu und führt selbst einige Berechnungen durch. Da Thread 2 die höhere Priorität als Thread 3 hat, wird Thread 3 verdrängt. Zum Zeitpunkt c kommt dann schließlich noch Thread 1 hinzu. Dieser will zum Zeitpunkt d ebenfalls die Semaphore sperren und blockiert sich dabei, weil Thread 3 die Semaphore hält. Da sich Thread 1 blockiert, kann Thread 2 weiterrechnen. Thread 2 beendet seine Berechnungen zum Zeitpunkt e und Thread 3 kann wieder weiterrechnen, weil Thread 1 immer noch auf die Semaphore wartet. Erst zum Zeitpunkt f gibt Thread 3 die Semaphore frei und Thread 1, der eigentlich die höchste Priorität hat, kann erst jetzt weiterrechnen.

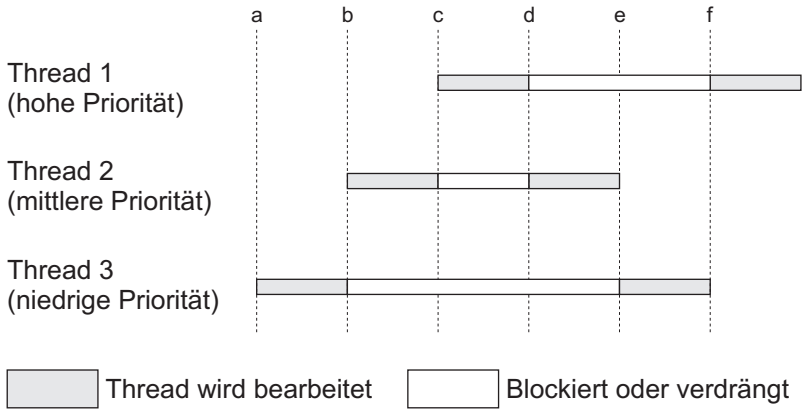

Abbildung B.1: Priority Inversion

Um das Problem der Prioritätsinversion zu vermeiden, wurde die Prioritätsvererbung implementiert. Abbildung B.2 zeigt das gleiche Beispiel wie Abbildung B.1. Thread 3 sperrt zum Zeitpunkt a die Semaphore und wird zum Zeitpunkt b von

Thread 2 verdrängt, welcher zum Zeitpunkt 3 wiederum von Thread 1 verdrängt wird. Dieser will zum Zeitpunkt d die Semaphore sperren und blockiert sich, da Thread 1 die Semaphore bereits gesperrt hat. Die Priorität von Thread 3 wird jetzt auf die Priorität von Thread 1 angehoben und Thread 3 kann somit sofort weiterrechnen, bis er zum Zeitpunkt e die Semaphore freigibt. Jetzt kann Thread 1 den kritischen Abschnitt betreten und seine Berechnungen durchführen und Thread 2 wird erst wieder zum Zeitpunkt f dem Prozessor zugeordnet.

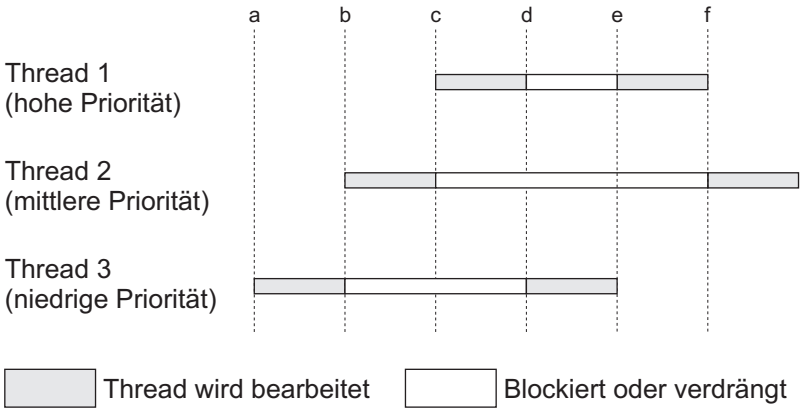

Abbildung B.2: Priority Inheritance

Bei der Durchführung einer P-Operation wird also überprüft, ob die Priorität des in die Warteschlange eingefügten Threads höher als die Priorität des Threads ist, der sich momentan im kritischen Abschnitt befindet. Ist das der Fall, dann wird dessen Priorität auf die höhere Priorität des wartenden Threads gesetzt.

Falls die Semaphore noch unbenutzt war, dann wird sie als benutzt markiert und der Thread kann den kritischen Abschnitt sofort betreten.

Der System-Call kehrt erst zurück, wenn der Thread den kritischen Abschnitt betreten kann. Dies wird dadurch erreicht, dass der System-Call auf eine Antwort vom Semaphoren-Manager wartet. Diese Nachricht wird jedoch verzögert, falls die Semaphore bereits benutzt wird.

Der System-Call liefert den Wert `SEM_OK` bei Erfolg zurück, bei einem Fehler den Wert `SEM_ERROR`.

B.2.3 V-Operation

Beim Verlassen des kritischen Abschnitts muss der Thread die Semaphore wieder freigeben. Dies geschieht durch den Aufruf des folgenden System-Calls:

```
SyscallError semV(unsigned int number);
```

Dieser System-Call wird als `SEM_V`-Nachricht an den Semaphoren-Manager geschickt, der dann die Funktion `semaphoreV` aufruft. Dort wird zunächst wieder überprüft, ob es sich um eine gültige Nummer handelt. Falls die Priorität des Threads zuvor hochgesetzt wurde, dann wird sie jetzt wieder auf den ursprünglichen Wert gesenkt. Anschließend wird überprüft, ob es noch wartende Threads gibt. Der erste Thread wird aus der Warteschlange entfernt und es wird eine Nachricht an ihn geschickt. Falls es wartende Threads gibt, wird auch hier sofort wieder überprüft, ob die Priorität des neuen Threads hochgesetzt werden muss.

B.3 Speicher-Management

Da es bei Topsy kein Mapping gibt und der TLB nur einmal initialisiert und dann nicht mehr verändert wird, ist die Größe des Benutzeradressraums auf 256 kB beschränkt. Um dieses Problem zu beseitigen, musste eine Seiten-Kachel-Tabelle erstellt werden, welche die Zuordnung von virtuellen zu physikalischen Seiten speichern kann. Da der Kernadressraum direkt auf den physikalischen Speicher abgebildet wird und somit die unteren 768 kB verwendet, kann für den Benutzeradressraum der restliche freie, physikalische Speicher verwendet werden.

Die Verwaltung der Seiten-Kachel-Tabelle, der Liste der freien und belegten Kacheln geschieht mit den folgenden Strukturen:

```
typedef struct Frame_t {
  unsigned int PFN; // physical frame number
  Address hint;
} Frame_Desc;

typedef struct Page_t {
  unsigned int PFN; // physical frame number
} Page_Desc;

HashList PFT; // page frame table
List freeFrames;
List usedFrames;
```

Die Variable `freeFrames` enthält die freien, `usedFrames` die belegten Kacheln. Die Seiten-Kachel-Tabelle wird in der Hashtabelle `PFT` gespeichert.

Das Mapping wird in der Funktion `mmInitMemoryMapping` in Memory/mips/MMDirectMapping.c initialisiert. In dieser Funktion werden die Listen der freien und belegten Kacheln, die Seiten-Kachel-Tabelle und der TLB initialisiert.

Auch die Funktionen `mmMapPages`, `mmUnmapPages` und `mmMovePage` in der Datei Memory/mips/MMDirectMapping.c wurden an die neue Situation angepasst.

Eine wichtige Änderung betraf die Funktion `mmUTLBError` in der Datei Memory/mips/MMError.c. Diese Funktion wird aufgerufen, wenn auf eine virtuelle Seite zugegriffen wird, die sich nicht im TLB befindet. Die Funktion durchsucht dann die Seiten-Kachel-Tabelle, ermittelt die zugehörige physikalische Seite und trägt die Informationen dann in den TLB ein.

Anhang C

Erweiterung des Simulators (LCD)

Um Topsy ein Display zur Verfügung zu stellen, wurde der Simulator MipsSim erweitert. Neu hinzugekommen ist die Datei LCD.java. Außerdem wurden einige Änderungen in Processor.java vorgenommen.

In Processor.java wird eine neue Speicherregion (40012 Bytes) für den Zugriff auf das Display eingefügt. In der Funktion `run` wird außerdem dafür gesorgt, dass das Display regelmäßig aktualisiert wird.

Der Simulator bietet ein 200×200 Pixel großes schwarz-weiß Display, welches in Topsy über einen Treiber angesprochen werden kann (IO/Drivers/lcd.c). Die Basisadresse des Displays ist `0xbff00000`. Dieser Speicherbereich ist insgesamt 40012 Bytes groß. Die ersten 12 Bytes werden für die Steuerung des Displays verwendet, über die letzten 40000 Bytes kann auf die einzelnen Pixel des Displays zugegriffen werden.

Durch das byteweise Schreiben der Struktur `lcdControl` an Adresse `0xbff00000` kann das Display gesteuert werden.

```
typedef enum {RECTANGLE=0, OVAL, PIXEL, LINE,
              ROUNDRECTANGLE} lcdAction;

typedef struct lcdControl_t {
  char ctl1;
  char action;
  unsigned short x1, y1, x2, y2;
  char color;
  char ctl2;
} lcdControl;
```

Momentan sind Funktionen zum Zeichnen eines Rechtecks, einer Ellipse, eines einzelnen Pixels, einer Linie und eines abgerundeten Rechtecks implementiert.

Die Variablen `ctl1` und `ctl2` können beliebige Werte annehmen. Diese Variablen werden nur dazu verwendet, um dem Simulator den Anfang und das Ende der Übertragung der Kontroll-Struktur zu signalisieren, da bei der Übertragung zuerst an Adresse `0xbff00000` und zuletzt an Adresse `0xbff0000b` geschrieben wird.

Die Variable `action` gibt die Funktion an, die auszuführen ist (Rechteck, Ellipse, ...).

Die Variablen `x1`, `y1`, `x2`, `y2` geben die Größe des zu zeichnenden Objekts an, wobei beim Zeichnen eines (abgerundeten) Rechtecks oder einer Ellipse die Variable `x2` die Breite und `y2` die Höhe angibt.

`color` gibt die Farbe an, in welcher das Objekt gezeichnet werden soll. Ein Wert von `0` steht dabei für die Farbe schwarz, für alle anderen Werte wird die Farbe weiß verwendet.

Über die Adressen `0xbff0000c` bis `0xbff9c4b` können die Pixel direkt gesetzt werden. Das Bild wird dabei zeilenweise auf den Speicher abgebildet. Eine Funktion zum Setzen eines Pixels könnte so aussehen:

```
void pixel(char x, char y, char color) {
  ThreadId lcd;
  char data[3];
  unsigned long int size=3;
  ioOpen(IO_LCD, &lcd);
  data [0] = x;
  data [1] = y;
  data [2] = color;
  ioWrite(lcd, data, &size);
  ioClose(lcd);
}
```

Den Rest übernimmt der Treiber mit folgender Funktion:

```
Error lcd_write(IODevice this, ThreadId threadId,
                char* buffer, long int* size) {
  char *address;
  if (*size == 3) { /* Direkter Zugriff auf die Pixel*/
    address = (char *) (12
        + ((unsigned long)(this->base) + buffer[0]
        + buffer[1]*200));
    *address = buffer[2];
    return IO_WRITEOK;
  }
}
```

Literaturverzeichnis

[Bre99] Thomas Bretscher. *Topsy – Teachable Operating System: Dynamischer Modul-Lader*. Computer Engineering and Networks Laboratory, ETH Zurich, 1999.

[Com88] Douglas E. Comer. *Operating System Design: The XINU Approach, Volume 1*. Prentice Hall, 1988.

[Fan] George Fankhauser. *Topsy – A Teachable Operating System*. http://www.tik.ee.ethz.ch/~topsy/Book/Topsy_1.1.pdf.

[GDB] *Debugging Workloads with gdb-simos*. http://simos.stanford.edu/userguide/userguide-29.html.

[Hau97] Franz Hauck. *Systemprogrammierung 1*. Institut für Mathematische Maschinen und Datenverarbeitung (IV), Friedrich-Alexander-Universität Erlangen-Nürnberg, 1997.

[Hof99] Fridolin Hofmann. *Betriebsprogrammierung 2*. Institut für Mathematische Maschinen und Datenverarbeitung (IV), Friedrich-Alexander-Universität Erlangen-Nürnberg, 1999.

[IDT94] Integrated Device Technology, Inc. *The IDTR3051, R3052 RISController Hardware User's Manual*, 1994. http://www.idt.com/docs/79R3051_MA_65105.pdf.

[Mos99] Dominik Moser. *TopsySMP – A Small Multi-Threaded Microkernel for Symmetrical Multiprocessing Hardware Architectures*. Computer Engineering and Networks Laboratory, ETH Zurich, 1999.

[Nac] *Nachos*. http://www.cs.washington.edu/homes/tom/nachos/index.html.

[Ruf98] Lukas Ruf. *Topsy i386 – A Teachable Operating System - The Port to the ia32 Architecture*. Computer Engineering and Networks Laboratory, ETH Zurich, 1998.

[Sch98] David Schweikert. *A lightweight and high-performance TCP/IP stack for Topsy.* Computer Engineering and Networks Laboratory, ETH Zurich, 1998.

[Sima] *The SimOS Home Page.* http://simos.stanford.edu/.

[Simb] *SPARC-binary of gdb-simos.* ftp://ftp.tik.ee.ethz.ch/pub/people/gfa/gdb-simos.

[SREa] *Motorola S-record file and record format.* http://www2.ele.ufes.br/ hans/68hc11/srec.html.

[SREb] *Motorola S-record description.* http://www.ndsu.nodak.edu/instruct/tareski/373f98/notes/srecord.htm.

[Tan90a] Andrew S. Tanenbaum. *Betriebssysteme - Entwurf und Realisierung, Teil 1.* Prentice-Hall International Inc., London, 1990.

[Tan90b] Andrew S. Tanenbaum. *Betriebssysteme - Entwurf und Realisierung, Teil 2.* Prentice-Hall International Inc., London, 1990.

[Tan99] Andrew S. Tanenbaum. *Computerarchitektur, 4. Auflage.* Prentice Hall, 1999.

Index