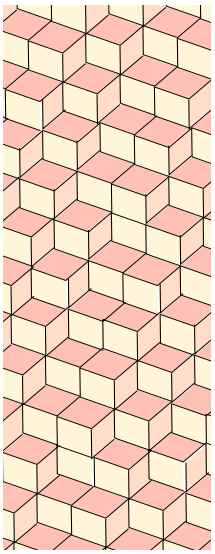
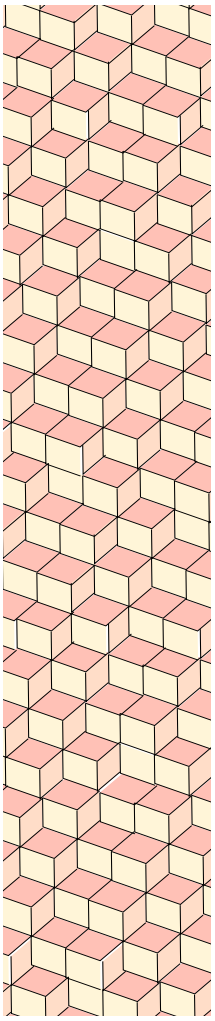


Topsy - A Teachable Operating System

Version 1.1, 20000322

George Fankhauser, Christian Conrad,
Eckart Zitzler, Bernhard Plattner

Computer Engineering and Networks Laboratory, ETH Zürich



This report describes the design and implementation of the Topsy operating system, a simple and tiny micro kernel for teaching purposes.

Hyperlinked source code and examples are available from the following webpage:
`http://www.tik.ee.ethz.ch/~topsy`

CONTENTS

1. INTRODUCTION

Topsy is a small operating system which has been designed for teaching purposes (Topsy stands for **T**eachable **OP**erating **SY**stem). It constitutes the framework for the practical exercises related to the course Computer Engineering II. This course is taught at the Department of Electrical Engineering at ETH Zürich and deals with the basic concepts of operating systems.

When planning and working out the lecture we thought about the best way how to teach this subject. In our opinion the concepts can best be learned by a combination of theory and practice, so we were looking for an operating system which serves as

1. a basis for practical exercises, enabling the students to apply the knowledge acquired in the lecture lessons, and
2. an example how basic principles can be implemented in a real operating system.

For us it was essential that the desired system improves the student's comprehension in the fields of process parallelism, communication and synchronization, interfacing hardware and memory management.

Though there is a myriad of operating systems (OS) for almost any hardware platform, none of these OSs fulfils the requirements we have, which are in particular:

- *Simplicity:* Powerful kernels tend to be too large and over-featured. Many algorithms are tuned for speed, lacking a simple formulation. Also, many unnecessary modules are found in kernels for performance reasons.
- *Readability:* OS code is often obscured due to its historical origin and lots of options and hardware variants being supported.
- *Hardware independence:* While most systems claim to be portable, we found that the effort of porting a complex existing system is comparable to building a new simple one.
- *Transparency:* For didactic reasons we favor clear program code instead of efficient realizations. The basic principles implemented in an OS have to be easily comprehensible by the students. Most OSs are designed for efficiency.
- *Lecture compatibility:* Since this is a course for students in the second academic year, we have to take into account what they have learned so far. One implication is the use of the C programming language because it was introduced in the lecture Computer Engineering I.

So we have decided to develop a new operating system from scratch. Our main goal was to design a small, easy to understand, well structured, but also realistic system. The focal point was explicitly on didactic interests. This report describes the design and implementation of Topsy. Design principles, algorithms, and data structures found in many other operating systems like Unix, Minix, Oberon, or Mach, only to name a few, were incorporated in Topsy.

Since this manual cannot replace a textbook about operating systems, we give references to the following book, whenever using technical terms that need further explanation:

> A. Silberschatz, P. Galvin, G. Gagne: Applied Operating System Concepts, 1st Edition, John Wiley & Sons, Inc., 2000.

We refer to it by the short cut AOSC (e.g. *see AOSC, chapter x, section x.yAOSC*).

Furthermore, we use a special sign throughout this book. Whenever you see this sign

at the border, you should be aware that the information you are reading is hardware dependent and related especially to the MIPS processor. Topsy was built with portability in mind, but when describing implementation details it is necessary to refer to the platform (IDT MIPS R3052E based board) used in the practical course.

2. Topsy in a Nutshell

This chapter gives an overview over the Topsy operating system. In particular, it is described what concepts were applied, what underlying system structure was chosen and what interface to user programs (system call interface) is provided.

2.1 Topsy Characteristics

When characterizing an operating system, normally there is a separation between the main tasks of an OS: process management, memory management and input/output interfacing.

2.1.1 Threads

To practice real situations with concurrent processes, experience race conditions, and to build and use synchronization primitives, it was mandatory to implement Topsy as a multi-threaded operating system (i.e. multiple threads are able to run quasi-parallel). Here we have to differ between the terms *process* and *thread*. A *process* can informally be described as a program in execution (see AOSC, chapter 4). It is defined by the resources it uses and by the location at which it is executing. In Topsy, there exist exactly two processes: the user process and the kernel process (operating system). *Threads*, however, share their resources, in particular they run in the same address space (for more information see AOSC, section 4.1.4). The user process as well as the kernel process contain several threads.

In Topsy all threads (of a specific process!) are running in one address space and may share global memory between them. Synchronization of shared memory is accomplished via *messages* (AOSC, section 4.5.1AOSC) between multiple threads. The private area of a thread is its stack which is not protected against (faulty) accesses from other threads. However, a simple stack checking mechanism has been incorporated to terminate threads on returning from their main function (stack underflow).

2.1.2 Memory

Topsy divides the memory into two address spaces: one for user threads and the other for the OS kernel (separation of user and kernel process). This has the advantage of better fault recognition facilities, and a stable and consistent behavior of the kernel (user threads are not able to crash the system by modifying kernel memory). The memory is organized in paged manner, i.e. the whole address space is split up into blocks of a predefined size (paging is explained in section 9.4, AOSC). Furthermore, the two address spaces are embedded in one *virtual address space*, although no swapping of pages to secondary memory is supported (see section 10.2, AAOSCOSC, for more details).

Topsy itself comes with a small footprint. It is able to run with a few 10 kilobytes of memory which is managed in a dynamic fashion. This ensures good utilization of memory. Threads can allocate memory by reserving a certain, connected piece of the virtual address space. We call these pieces consisting of several pages *virtual memory regions*. Every virtual memory region is assigned an appropriate number of physical pages (for the distinction between physical and virtual addresses refer to AOSC, section 9.1.2AOSC).

2.1.3 Input/Output

The Input/Output (I/O) subsystem of Topsy provides a framework for writing and installing hardware device drivers. A serial interface driver is implemented, as an example.

2.2 System Structure

To ensure that undergraduate students will be able to understand Topsy, it has a strict modular structure. The kernel contains three main modules reflecting the basic OS tasks: the memory manager, the thread manager and the I/O subsystem. All kernel modules are independently implemented as threads and therefore preemptable (i.e. they can be interrupted like user threads). The communication mechanism is the same as in user threads: by sending and receiving messages, interaction between the kernel components takes place. This provides quick response to interrupts and automatic synchronization between modules. A multi-threaded kernel has several advantages: Synchronization and context switching is done by the natural mechanism that is used anyway (no more kernel context stacks...), reuse and readability profit, and the kernel designer is encouraged to use fine grained concurrency.

The module structure is depicted in figure 1. Big boxes represent the main modules, inner boxes stand for submodules (or subsubmodules). The dotted lines indicate three different layers: the hardware abstraction layer (HAL), the hardware independent kernel components and at the top the modules running in user space.

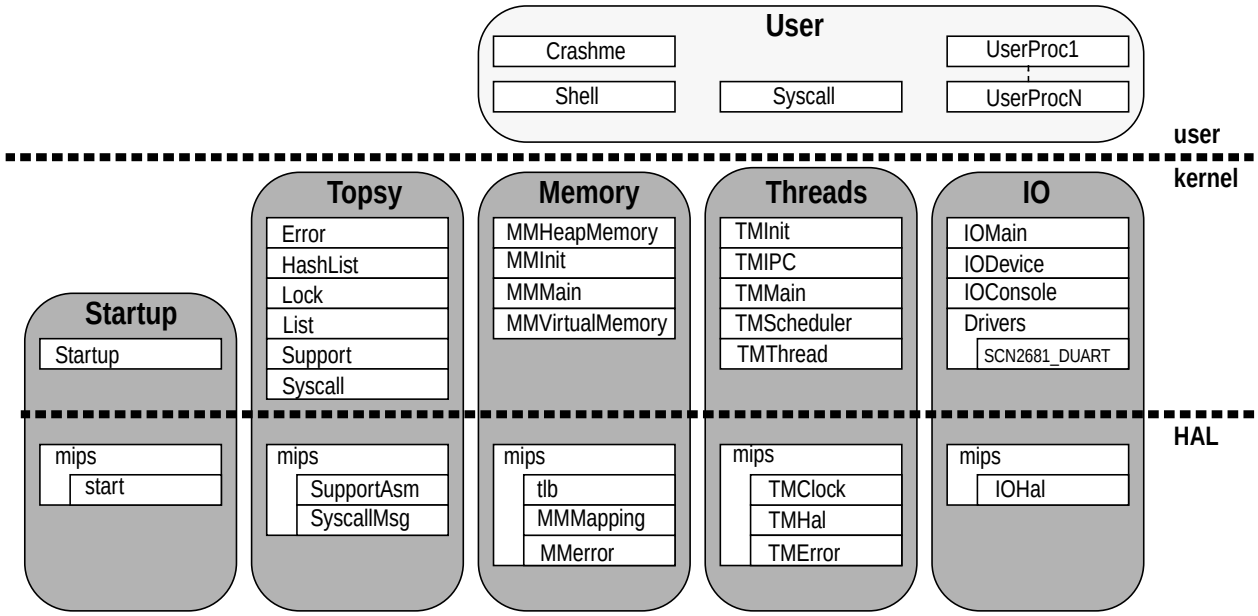


FIGURE 1. Modular structure of Topsy

The hardware abstraction layer is an important element of a portable design. Processor specific parts of the code like exception handling or context switching (to hand the control to another thread) are embedded in this layer. Consequently, HAL contains a few lines of assembler code. The two upper layers are completely written in C.

In the next subsections we give an overview over the particular modules.

2.2.1 Memory

The memory manager is responsible to execute the memory demands of kernel or user threads: allocating and deallocating virtual memory regions. This is primarily done in the module `MMVirtualMemory`. Receiving and sending messages as well as message dispatching is managed in `MMMain`, the initialization code for the memory manager is located in module `MMInit`. Furthermore, since virtual memory regions always consist of one or more pages (never two and a half and so on), espe-

cially for the kernel (and only for the kernel!) a second memory allocator with byte granularity is provided (`MMHeapMemory`).

The MIPS specific parts of the memory manager contain functions to manipulate the translation look-aside buffer (tlb, see AOSC, section 9.4.2.1AOSC) and to modify the mapping from virtual to physical addresses (`MMMapping`). `MMError` catches bus and address errors.

2.2.2 Threads

The modules `TMInit` and `TMMain` correspond to the ones in the module `Memory`. The first is responsible for proper initialization, the second acts as message dispatcher and contains the main function of the thread manager. Thread scheduling (AOSC, chapter 5) is done in `TMScheduler` which initially comes in a non-preemptive and a preemptive flavor (preemptive scheduling is explained in section 6.1.3, AOSC). Simple first-come, first-served scheduling (AOSC, section 6.3.1) resp. round-robing scheduling (AOSC, section 6.3.4AOSC) are used as scheduling algorithms. The interprocess communication (IPC), which is based on message passing, is realized in the module `TMIPC`. `TMThread` contains the implementation of system call functions provided by the thread manager.

In the hardware abstraction layer three modules are placed. `TMClock` is needed for preemptive scheduling and represents the interface to the hardware clock. `TMHal` on the other hand deals with exception and interrupt dispatching. `TMError` deals similar to `MMError` with catching specific exceptions.

2.2.3 IO

The I/O subsystem is structured in a very simple manner. `IOMain` covers the message dispatching (for opening and closing devices), `IODevice` represents a generic driver (a framework for device drivers) and `Drivers` contains the diverse driver modules (in the figure the driver for the serial interface is depicted). `IOConsole` provides simple output routines.

In `IOHal` functions to write to the console (needed for error or system messages) are located.

2.2.4 Topsy

The module `Topsy` can be seen as a collection of routines used by various kernel components. Informally spoken, it is the kernel library. Abstract data types like lists (`List`) and hash lists (`HashList`) are supported, also routines for printing error messages (`Error`). Furthermore, `Support` contains some small routines like copying memory blocks, `Lock` provides a simple locking mechanism (only for kernel use! - locks are described in section 7.4, AOSC). `Syscall` is the only module in `Topsy` which is used by user and kernel likewise. System calls are simple C functions exported by the `Syscall` module. But internally system calls are implemented by message passing which leads to a simpler implementation.

On the hardware dependent side there is a module containing the assembler code for the locking mechanism (`SupportAsm`) and a module realizing the two system calls for sending and receiving messages (`SyscallMsg`).

2.2.5 Startup

The start-up code is executed once to bootstrap Topsy (refer to section 13.3.2, AOSC). It is split up in a hardware independent (`Startup`) and a hardware dependent part (`start`).

2.2.6 User

At the top, the module `User` is located. A simple command line shell as well as the above mentioned syscall library (an identical copy) can be found here. Besides there is a little program `Crashme` which tries to break down the OS (a good test for stability!). Last but not least the user programs written by the students belong to the main module `User` (in the figure outlined as `UserProc1` to `UserProcN`).

2.3 System Call Interface

System calls (for short syscalls) provide the interface between a user thread and the Topsy operating system (refer AOSC, section 3.3) and are collected in the syscall library mentioned before. These library calls (implemented as C function calls) prepare the parameter to be passed, add the id of the system call and execute a software trap instruction (`syscall` on MIPS processors) to switch to kernel mode and branch to the code that services the system call. More precisely, a system call is mapped to a message which is delivered to the appropriate kernel module (for example `vmAlloc` is sent in a message to the memory manager). The user then waits (i.e. is suspended) for the answer (if any) to the system call.

In the following sections, a summary is given listing the system calls available to user programs (sorted by kernel module). The complete description together with the C prototype definitions can be found in the corresponding chapters (the appendix gives an overview, too).

2.3.1 Memory

In order to allocate memory, the system call `vmAlloc` is provided. The desired size is passed as a parameter, the address of the reserved virtual memory region is returned (by reference parameter). The counterpart, `vmFree`, deallocates a region which has been reserved earlier.

2.3.2 Threads

A new thread is created by the system call `tmStart`. Killing resp. finishing a thread can be accomplished by `tmKill` (a thread wants to kill another thread) or `tmExit` (a thread wants to finish its own execution). `If tmExit` is the last instruction in a user program it can be omitted, because `tmExit` is automatically called when a thread terminates. A very important function concerning non-preemptive scheduling is `tmYield`. It yields the control or CPU to another thread (if there is one ready to run), otherwise the calling thread regains the CPU (i.e. it continues execution). In the preemptive case, the CPU is automatically assigned to threads, depending on predefined time slices of control (in the next chapter this subject is described in more detail). Additionally, a system call for getting information about a thread is available (`tmGetInfo`).

For sending and receiving messages `tmMsgSend` resp. `tmMsgRecv` have to be called. `tmMsgSend` expects both the destination and the message itself as parameters, using `tmMsgRecv` it must be specified from which thread a message should be received, which kind of message is waited for and which amount of time should be maximally waited.

2.3.3 IO

Like most other operating systems, Topsy provides syscalls to open and close devices (`ioOpen` and `ioClose`). Reading from and writing to devices is accomplished by invocation of `ioRead` resp. `ioWrite`. Initializing a device for the first time is done via `ioInit`. For device control, special messages may be sent to certain device drivers.

2.4 Outline of the Report

Up to now the reader should have got an idea of what Topsy is and how it works. In the following chapters you will get more information about the internals and the

implementation. Firstly, the three main modules thread manager, memory manager and I/O subsystem are presented. Each module description begins with a formal definition of the corresponding system calls and goes afterwards more and more in detail. The whole bootstrapping process is treated in a single chapter. Here we will give answers to the questions how Topsy is loaded onto the MIPS board used in the practical course and how Topsy is started. Two further chapters deal with the user programs available (shell, crashme) and possible extensions of Topsy. In the appendix common data types and routines used by several kernel modules are described.

3. THREAD MANAGEMENT

3.1 System Calls

In Topsy all system calls are based on the exchange of messages between kernel and user threads. For most of them, a reply is expected from the kernel. This section first presents a high-level description of the message based IPC (interprocess communication, see AOSC, section 4.5) mechanism. Then, all system calls related to thread management are listed and explained.

The synopsis of both send and receive system calls is depicted below. Messages are described by the `Message` structure (located in Messages.h) which is formally given in the IPC section on page 14. The sender or receiver of a message is specified by an identifier (id) of type `ThreadId` (see appendix). All threads get a different id when they are created.

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

The sending of messages is nonblocking and returns either `TM_MSGSENDOK` or `TM_MSGSENDFAILED` for an error.

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

The receive message function attempts to read a message during at most `timeOut` micro-seconds. If `timeOut` is set to `INFINITY`, then the function is blocking, other timeouts are not implemented yet and therefore not allowed. A thread identifier set to `ANY` allows receiving of messages from any sender, on return, `*from` contains the sender's thread identifier. As second parameter, the kind of message which is expected must be specified; by passing `ANYMSGTYPE` the next arriving message (no matter which type) is taken. The received message is pointed to by `msg`. `tmMsgRecv` returns `TM_MSGRECVFAILED` in the case of an error, on success `TM_MSGRECVOK` is returned.

3.1.1 tmStart

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

A new user thread is created via the `tmStart` system call. It returns either `TM_STARTOK` when the thread has been properly created or `TM_STARTFAILED` in case of failure. The second parameter is a pointer to the function that has to be started as a thread. It is possible to pass a single parameter to a new thread via the third argument of `tmStart`. The fourth parameter is a logical name that is given to the new thread. On return, `*id` (the first parameter) contains the id of the new thread (return value by reference).

3.1.2 tmExit

```
void tmExit(void);
```

In order to complete its execution and free resources, a thread has to invoke `tmExit`. It must be noted, however, that a call to `tmExit` is implicitly done when the return statement in the main function of the thread is executed. This is a protection mechanism to avoid stack underflows. Hence, if the call to `tmExit` is the last instruction in the main procedure of a thread, it can be omitted.

3.1.3 tmYield

```
void tmYield(void);
```

With a non-preemptive scheduler, a user thread can yield the CPU for the benefit of another thread (cooperative scheduling). However, if no other thread can be scheduled, it continues its execution. This function may also be used in the preemptive case to force a scheduling decision.

3.1.4 tmKill

```
void tmKill(ThreadId id);
```

A user thread is allowed to kill another user thread (given by the parameter of `tmKill`). On the other hand, if a user thread tries to kill a kernel thread it is killed itself. For kernel threads there is no restriction, they may kill all kinds of threads.

3.1.5 tmGetInfo tmGetFirst tmGetNext tmGetThreadByName

The following group of system calls exchange information about threads or do some lookups. They are implemented by using the same system message which is subtyped.

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

By invoking `tmGetInfo` one gets information about a particular thread (first parameter `about`). In order to get its own id, a thread can pass `SELF` as first parameter.

By calling `tmGetFirst` and `tmGetNext`, a user program can enumerate all running threads on the system. A structure is filled with the id, parent id, status and name. For example, the shell command “ps” is implemented using these system calls.

The system call `tmGetThreadByName` searches for a thread called `name`. This assumes that a string was passed as name argument when the thread was started.

3.1.6 tmGetTime

This system call is used to get the time of day. It is returned as seconds and micro seconds since January 1st, 1998, 00:00, UTC. All presentation and time zone issues must be handled by the user. The resolution of the clock depends on the time slice defined for the system. On the MIPS implementation this is 10 ms.

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

3.1.7 tmSetTime

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

`tmSetTime` is used on systems which do not have a battery backed real time clock or to adjust the current time.

3.2 Interprocess Communication (IPC)

A system call causes a message to be created and sent to the kernel via a software interrupt mechanism (the MIPS processor provides the `syscall` instruction for that - refer to AOSC, section 3.3). The exact structure of these messages is introduced in the next subsection. Each system call is recognized via an identifier in the message structure. The list of these identifiers for the thread manager component can

be found in Messages.h (see definition of type `MessageId`). A dedicated function in the kernel (`msgDispatcher` in file TMIPC.c) processes syscall exceptions and copies the arriving message to the message queue of the destination thread.

There is a single IPC mechanism in Topsy, namely the sending/receiving of messages (interprocess communication is explained in section 4.5, AOSC). Messages can be sent either between user and kernel threads, between kernel threads, or between user threads (e.g., for synchronization purposes). Both kernel and user threads have a unique fixed sized queue for storing incoming messages (managed by a FIFO policy, i.e., first in, first out). Each thread possesses one queue. The size of this queue is a constant `MAXNBMSGINQUEUE` (defined in Configuration.h), and an attempt to send a message to a thread with an already full queue causes the send operation to fail with `TM_MSGSENDFAILED`.

3.2.1 Messages and Message Queues

The relevant message structures used in the thread manager are exposed in the following lines (for details refer to Messages.h). A generic message type (`Message`) is created with the sender id (set by the kernel for security reasons) and the type of the message (set by the sender). The third field is a union of all possible messages that can be exchanged in Topsy. If two user threads want to communicate, the specific message structure of type `UserMessage` has to be used (the message id can be arbitrarily chosen).

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

/* example for a specific message structure */
typedef struct TMStartMsg_t { /* Message to create a thread */
    ThreadMainFunction fctnAddress;
    ThreadArg parameter;
    char *name;
} TMStartMsg;

typedef union SpecMsg_u {
    UserMessage userMsg;
    TMStartMsg tmStart; ...
    VMAllocMsg vmAlloc; ...
    IOOpenMsg ioOpen; ...
    SyscallReply syscallReply;
} SpecMessage;

typedef struct Message_t {    /* Main message structure */
    ThreadId from;            /* Sender of the message */
    MessageId id;             /* Type of the message */
    SpecMessage msg;          /* Specific message contents */
} Message;
```

In Topsy, each kernel and user thread has its own message queue, where incoming messages are stored. The way messages are inserted in the queue (and then extracted) is depending on the adopted policy. The simplest one is FIFO (first in, first out), where all packets are retrieved in the same sequence as they arrived. However, this simple policy turned out to be insufficient, and even error-prone, for the purposes of the Topsy message-passing mechanism. Therefore, a mix of FIFO and pri-

ority queuing was adopted in the way described in Figure 2 on page 16. The message

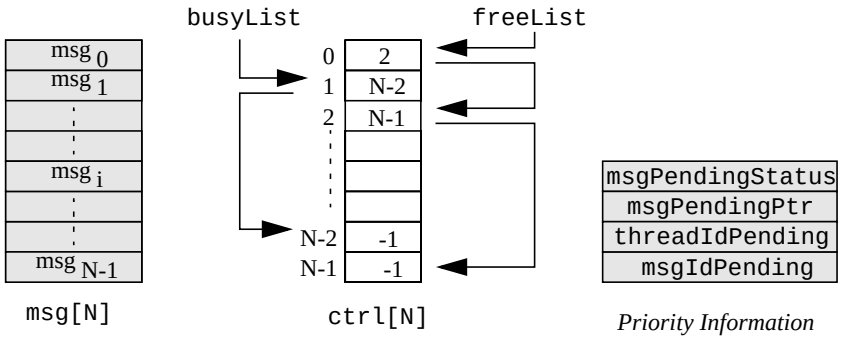


FIGURE 2. Message Queue Structure

queue is based on a fixed-size array of messages (`msg[N]`). The linking of messages inside this array is realized by two integers, namely `busyList` and `freeList`, which refer to another fixed-size array of integers (`ctrl[]`). The value found in the second array indicates at which position the next message is to be found. A value of `-1` in the `ctrl[]` array means that this is the last message of the list.

To implement a mix of FIFO and priority queuing, an additional structure is used. Within the four fields given on the right of Figure 2 on page 16, a thread has the possibility to request a particular message of type `msgIdPending` coming from thread `threadIdPending`. If such a message already exists in the `msg[]` queue, it is directly delivered to the thread. In the other case, the receiving thread is blocked and the next time it is scheduled, either the required information will have been copied at the address `msgPendingPtr`, or a `TM_MSGRECVFAILED` will have been returned. The `msgPendingStatus` is an internal flag for the implementation of a very simple finite state machine.

The rationale for this priorization of messages is the need of threads to be sure to receive certain important messages. For example, when a *syscall* is made the calling thread usually waits for a reply carrying the return parameters. However, it cannot be guaranteed that this reply is put into the first place of the queue, it may be possible that the calling thread receives a message from another thread before the reply to the syscall arrives. To prevent infinite waiting, messages that are expected by sender id

or message type are bypassed even when there are other, not expected messages in the queue.

USER

```
tmMsgSend:
    sw a0, 0(a1)  /* dest. thread id put into msg from */
                                                /* field */
    li a3, SYSCALL_SEND_OP
    /* Argument registers are now:
        a0: - (undefined)
        a1: message reference (Message*)
            (destination thread id included)
        a2: - (undefined)
        a3: SYSCALL_SEND_OP
    */
    /* Raising of an exception (trap to kernel mode) */
    syscall
```

KERNEL (exception handling in msgDispatcher)

```
to = msgPtr->from;
msgPtr->from = fromId;/* copy sender id to msg */
if (kSend(to, msgPtr) != TM_OK ) {
    tmSetReturnValue(threadPtr->contextPtr,
    TM_MSGSENDFAILED);
} else {
    /* message was successfully copied in messageQueue*/
    tmSetReturnValue(threadPtr->contextPtr, TM_MSGSENDOK);
}
/* A new scheduling decision was taken in kSend(), as
 * kSend() may be called from other modules, independently
 * of msgDispatcher()
```

FIGURE 3. Code fragment for message sending

3.2.2 Sending a Message

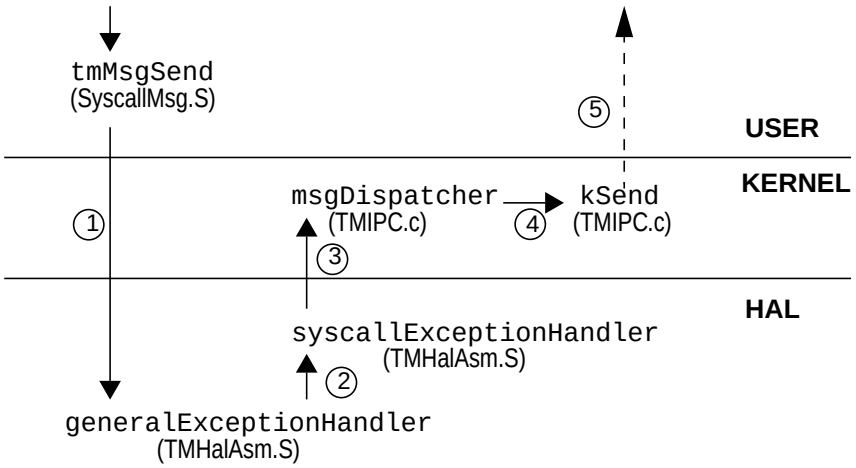
One has to distinguish between the `tmMsgSend` and the kernel `kSend` function. The `tmMsgSend` call issues an exception that is caught in the corresponding exception handler in the kernel. This is the function to call each time a message has to be sent from any thread to any other. Inside the exception handler, the message is actually copied in the destination's message queue via the `kSend` function. The `kSend` function can only be invoked inside the kernel (no external visibility).

```
int kSend(ThreadId dest, Msg *msg);
```

If the receiver's message queue is full, or the destination thread does not exist, `kSend` fails. On success, `kSend` returns `TM_OK`. Note that `kSend` causes a new scheduling decision if necessary. The structure of the message queue is given below (see tm-include.h):

```
typedef struct MessageQueue_t {
    int freeList; /* index of first free cell */
    int busyList; /* index of first valid message */
    int ctrl[MAXNBMSGINQUEUE]; /* order of messages */
    Message msg[MAXNBMSGINQUEUE]; /* message array */
```

```
    PendingStatus msgPendingStatus;
    Message* msgPendingPtr;
    ThreadId threadIdPending;
    MessageId msgIdPending;
} MessageQueue;
```



1. a syscall instruction causes an exception
2. the general exception handler branches to the syscall exception handler
3. decision whether to send or receive a message
4. the message is copied in the destination queue
5. a scheduling decision is done and CPU is yielded to the chosen thread

FIGURE 4. Control flow for message sending

An in-depth description of the message sending process is provided in Figure 3 on page 17 (code fragments). The control flow, i.e. the sequence of internal function calls, is depicted in Figure 4 on page 18.

3.2.3 Receiving a Message

A message may be received via the `tmMsgRecv` function. As for the sending of messages, a software exception is raised and the kernel checks whether a message is already available or not. If there is no message, the caller is put to sleep. On the other hand, if there is a message, it is read with the `kRecv` function. The kernel knows if messages are pending by checking for a field `msgPendingStatus` in the message queue structure of the corresponding thread.

```
int kRecv(ThreadId *from, Msg *msg);
```

The `from` parameter contains the thread identifier of the message sender, whereas the message is contained in the second parameter `msg`. As for the sending message process, a finer example is provided to illustrate the receiving of messages in Figure 5 on page 19 (code fragments). The control flow is almost identical to the

sending process. Obviously, at the top `tmMsgRecv` is called and at the end of the calling chain `kRecv` is invoked.

USER

```
tmMsgRecv:
    lw t0, 0(a0)     /* dereference a0 -> t0 */
    sw t0, 0(a2)     /* threadId copied into from field */
    sw a1, 4(a2)     /* msgIdcopied into id field of msg */
    move a1, a2        /* shifting of arguments */
    move a2, a3        /* shifting of arguments */
    li a3, SYSCALL_RECV_OP
    /* Argument registers are now:
        a0: - (undefined) (actually unchanged)
        a1: message reference (Message*)
        a2: timeout (unsigned long int)
        a3: SYSCALL_RECV_OP
    */
    /* Raising of an exception (trap) */
    syscall
    /* from field of tmMsgRecv() is set with */
    /*msgPtr->from */
    lw t0, MESSAGEFROM_OFFSET(a1)
    sw t0, 0(a0)
    /* The return value of tmMsgRecv() is in v0 */
```

KERNEL (exception handling in msgDispatcher)

```
/* Check if the expected threadId exists */
if ((hashListGet(threadHashList,
    (void**)(&fromThreadPtr),msgPtr->from) != HASHOK) &&
    (msgPtr->from != ANY)) {
    tmSetReturnValue(threadPtr->contextPtr,
        TM_MSGRECVFAILED);
} else {
    if (kRecv( msgPtr, threadPtr) != TM_OK ) {
        /* No corresponding message found, put receiver to
         * sleep, and make a new scheduling decision
         */
        schedulerSetBlocked(threadPtr);
        schedule();
    }
    /* The expected message was copied in msgPtr */
    tmSetReturnValue(threadPtr->contextPtr, TM_MSGRECVOK);
}
```

FIGURE 5. Code fragment for message receiving

3.3 Design Overview

In this section, we give an overview of the internal logical structure of the thread manager. There are two main tasks concerning thread management: interprocess communication (IPC) and CPU scheduling (see chapter 5, AOSC). The IPC mechanism is implemented in the module `TMIPC` (described in the previous section), while CPU scheduling is realized by `TMScheduler` (section 5.5 on page 27). Handling exceptions and interrupts is detailed in the section “Hardware Abstraction Layer (HAL)” on page 28. Furthermore, three main modules have to be mentioned here: `TMInit`, `TMMain`, and `TMThread`. `TMInit` concerns the initialization pro-

cess of the thread manager which is described in section 3.7 on page 27. Message dispatching (meaning interpreting messages and arranging the corresponding actions) is done in `TMMain`. The implementation of the thread manager syscalls can be found in `TMThread` (see section 3.4 on page 20).

All modules share the data structures containing information about threads (see definition of data type `Thread` in section 3.6 on page 26). Since every thread is identified by a certain integer, there is a list of all thread ids which stores references to the `Thread` structures. To allow quick access to specific thread data identified by a thread id, additionally, a hash list of all thread ids is maintained.

3.4 Processing Syscalls

This section deals with the implementation of the system calls provided by the thread manager. The functions described below are located in the module `TMThread` (files TMThread.h and TMThread.c).

3.4.1 Creating a Thread

When a `tmStart` system call message is recognized by the main message dispatcher (function `tmMain` in file TMMain.c), the message is decoded followed by a call to `threadStart` (file TMThread.c). This function has six arguments: the address of the function to be launched, a parameter passed to the new thread, the information whether a user or kernel is to be created (the type `AddressSpace` is defined in the memory manager), the thread name, the id of the parent thread and a flag indicating whether a lightweight thread is to be created. If the last parameter is set to FALSE, then a thread with minimal resources (without stack) is started (for example, the idle thread is a lightweight thread).

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

The following steps are performed in `threadStart`:

- Firstly, the thread manager allocates a new thread data structure via `hmAlloc` (see memory manager) within the dynamic kernel memory.
- In the next step, a `vmAlloc` call to the memory manager is performed to obtain an execution stack for the new thread. The initial size of the stack is a constant `TM_DEFAULTTHREADSTACKSIZE` (can be redefined if necessary). The stack is allocated in the kernel address space.
- A new thread id is generated and inserted in the thread id list (variable `threadList`) and the thread id hash list (variable `threadHashList`). The thread identifier of user threads is positive whereas it is negative for kernel threads (predefined values are `TMTHREADID` (-2) for thread manager, `MMTHREADID` (-1) for memory manager, `IOTHREADID` (-3) for the IO subsystem, and 2 for the main procedure of the user code).
- Afterwards the stack is moved to the user address space if a user thread is to be created.
- On success, the new thread data structure is updated with the thread identifier, the PC (program counter), and the SP (stack pointer). The complete thread context is initialized.
- The top of the new stack is laid out in a predefined manner (detailed in Figure 6 on page 22). The upper stack words are necessary to perform a `tmExit` call automatically when the thread terminates. The way `tmExit` is called after thread completion is described in the following section.

- The status of the new thread is set to READY (runnable).

3.4.2 Exiting a Thread

The function `threadExit` performs the termination of a thread. This is done in several steps:

- The thread identifier is removed from the thread id list and the thread id hash list.
- By invocation of `schedulerRemove` (TMScheduler.c), the scheduler is informed about the completion of the thread.
- All former reserved memory resources are deallocated (stack, thread descriptor, and allocated memory).
- All threads waiting for a message of the exiting thread are notified, i.e. the `tmMsgRecv` call returns `TM_MSGRECVFAILED`.

The mechanism how `tmExit` is automatically called when a thread completes is shown in Figure 6 on page 22.

This mechanism is a bit tricky. The `ra` register of the MIPS processor contains the return address (by convention). By setting this register to the beginning of a special exit code lying on the stack, this exit code is executed whenever a return (`jr ra`) in the program code (of the corresponding thread) occurs. Initializing the `ra` register is done in `tmStart`.

Consider a situation where a thread returns by the instruction `jr ra`. Afterwards, the PC points to the start of the exit code on the stack (remember that the stack grows downwards, while program code is executed from lower to higher addresses). The first five instructions compute the start address (stored in register `a1`) of the exit message located above the exit code. This message is a `tmExit` message. The next two instructions are identical to the last ones in `tmMsgSend`. Hence, what is done in the exit code is to send a `tmExit` message to the thread manager. This properly exits the thread.

3.4.3 Killing a Thread

Inside the thread manager, the function `tmKill` is invoked when a `TM_KILL` message is recognized:

```
Error threadKill(ThreadId killedId, ThreadId killerId);
```

After checking whether the kill request is allowed (user threads must not kill kernel threads) the same steps as described for `tmExit` are performed.

3.4.4 Yielding of a Thread

When a user thread wishes to yield the CPU, it invokes the function `tmYield`. In the kernel, a call to `threadYield` is then performed.

```
void threadYield(void);
```

If there is another user thread ready, a scheduling decision is forced (by a call to `schedule` in TMScheduler.c). If no other user thread can be scheduled, the requesting thread can continue up to the next `threadYield` call or until it is preempted by another thread with higher priority (e.g., a kernel thread). More details are provided in the scheduling section.

Note: this function is executed in exception context, i.e. when the low level message dispatcher (`msgDispatcher` in TMIPC.c) handles a `tmMsgSend` syscall exception it catches yield messages (message id is `TM_YIELD`) and invokes directly `threadYield`. Messages with id `TM_YIELD` are not delivered to the thread man-

ager message queue and therefore not dispatched in TMMain by the function `tmMain`. The reason for this special treatment of `tmYield` lies in the necessity of context switching (AOSC, section 4.2.3) after a scheduling decision. Since context switching can only be done in exception mode, `threadYield` also has to be executed in exception mode. Because the semantics of yield is to give up the CPU immediately, the scheduling decision is made in the exception code.

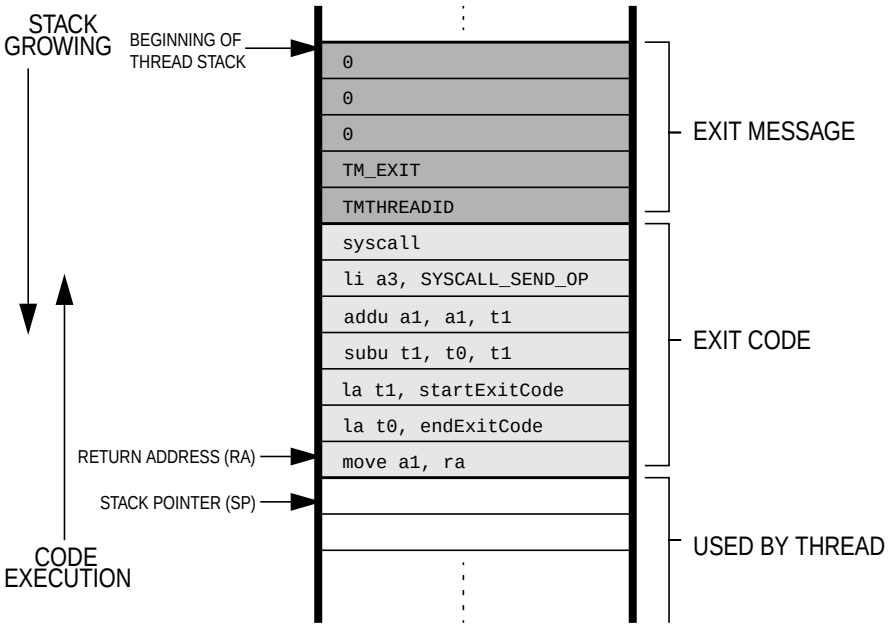


FIGURE 6. Stack processing for tmExit

3.4.5 Information about a Thread

Internally, the `tmGetInfo` syscall is implemented by the function `threadInfo`:

```
Error threadInfo(ThreadId id, ThreadId aboutId,
    ThreadInfoKind info, ThreadInfo* infoPtr,
    long int values[]);
```

3.5 Scheduler Design

The Topsy scheduler uses a multilevel queueing system (AOSC, section 6.3.5) with three levels, corresponding to kernel, user, and idle threads. Within each level, a round robin policy (see AOSC, section 6.3.4) is used. The highest priority is devoted to kernel threads, i.e., no user thread may be scheduled if there is a single runnable kernel thread. Idle threads have lowest priority (what a idle thread is and what it is good for is explained below).

Each thread may be in one of the following states:

- RUNNING: the thread is currently running (at any time, only a single thread may be in the RUNNING state).
- READY: the thread is ready to be scheduled and then dispatched.
- BLOCKED: the thread is blocked, e.g., waiting to receive a message.

Transitions are only allowed from RUNNING to either READY or BLOCKED, from BLOCKED to READY and from READY to RUNNING (see Figure 7 on page 23).

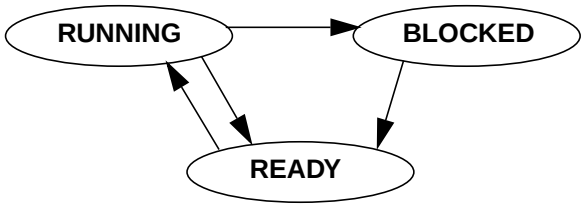


FIGURE 7. Thread state transition diagram

Both kernel and user threads are preemptive (idle threads, too). A new scheduling decision (schedule and dispatch of a new thread) may be performed in the following cases:

- Time quantum elapsed for the thread (if time-sharing mode).
- An exception is processed in the kernel. It can happen that a higher priority kernel thread has to preempt a running user thread (especially for system call processing). Hardware interrupts do not lead to a new scheduling decision.
- A running thread is put to sleep waiting for a message.
- After sending a message.

As mentioned before, there is a third kind of thread, namely idle threads. Only one idle thread exists, it is always ready (never blocked). The code for this thread is very simple, only an endless loop is performed. We need an idle thread in order to simplify the implementation. When all threads (user and kernel threads) are blocked, i.e. none is runnable, the question arises what should be done. The scheduler has to wait until an hardware interrupt wakes up an arbitrary thread. To avoid this situation, we introduced the idle thread which is made runnable whenever no other runnable thread exists.

How multilevel queue scheduling is implemented is shown in Figure 8 on page 24. There are three levels according to kernel, user, and idle threads. Zero stands for highest priority. For every priority level we have two queues: a READY queue and a BLOCKED queue. Since only one thread is able to run at a certain point of time, a running queue is not necessary.

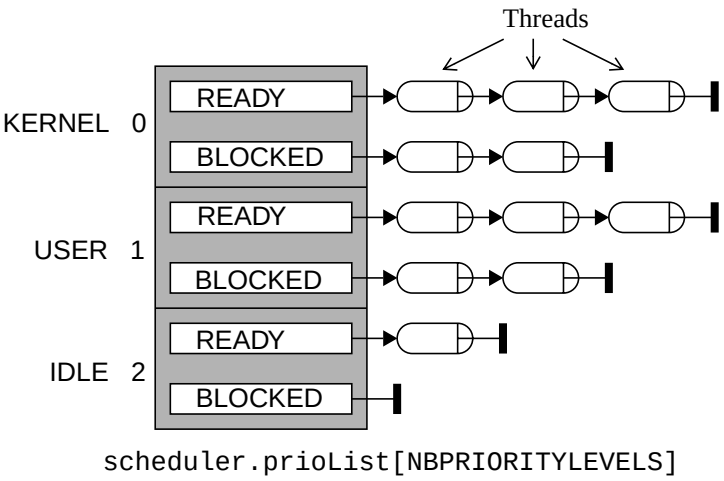


FIGURE 8. Multilevel queue scheduling (data structure)

3.5.1 Scheduler Interface

The internal behavior of the scheduler can be seen in four interface functions, which are `schedulerSetReady`, `schedulerSetBlocked`, `schedulerRunning` and `schedule`.

```
Error schedulerSetReady(Thread* threadPtr);
```

A thread is ready to be scheduled (BLOCKED - READY transition) and is moved from the BLOCKED queue to the end of the READY queue (according to the priority level of the thread).

```
Error schedulerSetBlocked(Thread* threadPtr);
```

A thread is put to sleep at the end of the scheduler BLOCKED queue (e.g., waiting for a message) via the `schedulerSetBlocked` call.

```
Thread* schedulerRunning(void);
```

The function `schedulerRunning` returns a pointer to the `Thread` structure of the current running thread.

```
void schedule(void);
```

The next ready thread with the highest priority is picked to be made runnable via the `schedule` call. At any time, only a single thread can be in RUNNING state. The actual context switch (saving and restoring of context) necessary after each `schedule` call is done via the HAL functions `saveContext` and `restoreContext` (located in TMHalAsm.S).

```
void saveContext(ProcContextPtr contextPtr);
void restoreContext(ProcContextPtr contextPtr);
```

Each thread has its own context which is defined by the register values at every point in time. When a thread is set from state RUNNING to BLOCKED or READY, the thread context (register values) must be saved (`saveContext`) in order to guarantee correct execution for the next time. The `restoreContext` function (which is the counterpart to `saveContext`) sets all register to the values that have been stored before. A `schedule` call is always followed by a `restoreContext` call

as a new thread becomes runnable. Thus, the `restoreContext` function ends with a jump instruction to the PC of the new restored thread context.

Note that the `schedule` function may run in an interrupt handler (e.g. the clock interrupt handler is realized by a simple call to `schedule`). Therefore spinlocks (see appendix and AOSC, section 7.4) are used to handle race conditions. The code for schedule is shown in Figure 9 on page 25.

```
void schedule()
{
    ThreadPriority priority;
    Thread* newRunning = NULL;
    Thread* oldRunning = scheduler.running;

    /* Test for locks... */

    /* Current running thread is put at the end of its ready queue,
     * unless it was previously put to sleep (by doing a msgRecv)
     */
    if (oldRunning->schedInfo.status != BLOCKED) {
      listMoveToEnd(
                scheduler.prioList[oldRunning->schedInfo.priority].ready,
                oldRunning, oldRunning->schedInfo.hint);
     oldRunning->schedInfo.status = READY;
    }

    /* Loop over all ready queues from higher to lower priority to find
     * next thread that is allowed to run
     */
    for (priority = KERNEL_PRIORITY; priority < NBPRIORITYLEVELS;
         priority++) {
      listGetFirst(scheduler.prioList[priority].ready,
          (void**)&newRunning);
      if (newRunning != NULL) {
          /* yup, we found a non-empty ready queue and we always pick
           * the first one. we already put the old running thread to its
           * queue end which results in a fine priority round-robin
           * scheduling.
           */
          newRunning->schedInfo.status = RUNNING;
          ipcResetPendingFlag(newRunning);
          break;
      }
      /* the idle thread(s) guarantee that we always find a ready thread */
    }
    scheduler.running = newRunning;
}
```

FIGURE 9. The scheduling function

3.5.2 Scheduler Internals

In the current design, there are three priority levels, which are kernel, user, and idle. To keep the design as general as possible and to allow future extensions, a generic model with maximum `NBPRIORITYLEVELS` priority levels is implemented. For each priority level, there are two thread lists (the abstract data type `List` is described in the appendix). The first one contains all threads in READY status, and the second all threads in BLOCKED status. The definition of the programming structures is provided below:

```
/* Thread priority level (must match NBPRIORITYLEVELS)
   Decreasing order of priority */
typedef enum {KERNEL_PRIORITY, USER_PRIORITY,
    IDLE_PRIORITY } ThreadPriority;

/* Two queues for each priority level */
typedef struct SchedPriority_t {
```

```
        List ready;
        List blocked;
    } SchedPriority;

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

The status of a thread is defined by:

```
    /* Thread status in scheduler */
    typedef enum {RUNNING, READY, BLOCKED} SchedStatus;
```

3.6 Internal Data Structures

The main data structure is the thread descriptor. It contains two parts, a context part and a control part with software information. The second part contains the identifier of the thread (`threadId`), its scheduling status (RUNNING, BLOCKED, or READY), the message queue, statistics, and so on (see the comments in the type declaration). On the other hand, the context part is extremely dependent of the processor. To facilitate porting to different platforms, an attempt was done to build a generic processor context with abstract names.

The C-structure is given below:

```
/* Processor dependent thread information */
typedef struct ProcContext_t {
    Register returnValue[2];       /* v0 - v1 */
    Register argument[4];          /* a0 - a3 */
    Register callerSaved[10];      /* t0 - t9 */
    Register calleeSaved[8];       /* s0 - s7 */
    Register globalPointer;        /* gp */
    Register stackPointer;         /* sp */
    Register framePointer;         /* fp */
    Register returnAddress;        /* ra */
    Register programCounter;       /* pc */
    Register hilo[2];              /* hi, lo registers */
    Register statusRegister;       /* sr after trapping */
} ProcContext;

/* Statistics about threads that may be gathered */
typedef struct ThreadStatistics_t {
    int cpuCycles;
} ThreadStatistics;

/* Scheduler specific data in each Thread structure */
typedef struct SchedulerInfo_t {
    SchedStatus status;    /* BLOCKED, READY, RUNNING */
    ThreadPriority priority;/* Priority of the thread */
    Address hint;          /* Backwards reference on list */
                           /* descriptor */
} SchedulerInfo;
```

```
/* Main thread structure */
typedef struct Thread_t {
    ProcContextPtr contextPtr;/* Processor dependent context */
    ThreadId id;              /* Processor independent */
    char name[MAXNAMESIZE];   /* Logical name of the thread */
    ThreadId parentId;        /* Thread id of parent thread */
    Address stackStart;       /* Start of stack */
    Address stackEnd;         /* End of stack */
    MessageQueue msgQueue;    /* Message queue */
    SchedulerInfo schedInfo;  /* Scheduler specific data */
    ThreadStatistics stat;    /* Various statistics */
} Thread;
```

3.7 Init Procedure

The init procedure for the thread manager is called by the start-up procedure of Topsy. The format of `tmInit` (file TMInit.c) is as follows:

```
void tmInit(Address mmStack, Address tmStack,
    Address userInit);
```

Both `mmStack` and `tmStack` were already initialized in `mmInit`, the initialization procedure of the memory manager. The last parameter `userInit` is the location of the main procedure of user code (as statically specified during the link process).

The following steps have to be performed in the `tmInit` procedure:

- Two thread structures for both memory and thread managers are used from static memory. The thread identifiers are predefined as -1 for the memory manager and -2 for the thread manager.
- Both stacks are prepared in the way shown in Figure 6 on page 22. This is done because it leads to shorter program code; although it is not necessary for these two kernel threads.
- The contents of both thread structures are updated, this includes the setting of the context (register values).
- Both threads are added to the common thread id list and the thread id hash list. These two data structures have to be initialized before.
- Specific exception handlers are installed in order to kill faulty threads (see section 3.8.1 on page 28).
- Both threads are set to READY.
- A scheduling decision is forced by a call to `schedule`.
- The system clock is configured in the right mode and the clock interrupt handler is installed.
- A context switch takes places. By calling `restoreContext`, the control is given to the first kernel thread. The initialization process is finished.

3.7.1 Starting the Thread Manager

The thread manager thread starts its work at `tmMain` (in file TMMain.c):

```
void tmMain(ThreadArg userInitAddress);
```

The `userInitAddress` parameter is the start address of the user code. Following steps have to be performed in `tmMain`:

- A new thread for input/output is created via `threadStart` (file TMThread.c).
- Then, the first user thread is started. The address of the init code was given as parameter to `tmMain`.

- The idle thread with lowest priority is generated.
- Now, the thread manager may enter its main loop, waiting for messages.

3.8 Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer mainly deals with the exception/interrupt processing and with the system clock configuration.

3.8.1 Interrupt/Exception Processing

The definition of the technical terms *exception* and *interrupt* (interrupts are introduced in AOSC section 2.1) slightly differs between various processors. To describe the exception model of the MIPS processor (R3051 architecture) we give an excerpt from the MIPS hardware user's manual (The IDTR3051, R3052, RISController Hardware User's Manual, Revision 1.4, July 15, 1994, Integrated Device Technology, Inc., page 5-1):

> The exception processing capability of the R3051 is provided to assure an orderly transfer of control from an executing program to kernel. Exceptions may be broadly divided into two categories: they can be caused by an instruction or instruction sequence, including an unusual condition arising during its execution; or can be caused by external events such as interrupts. When an R3051/52 detects an exception, the normal sequence of instruction flow is suspended; the processor is forced to kernel mode where it can respond to the abnormal or asynchronous event. [...]

Fifteen exceptions are recognized, ranging from Reset to Bus Error and Address Error, etc. All types are listed in the type declaration of `ExceptionId` (without Reset which is treated in a special way):

```
typedef enum {
        HARDWARE_INT,
        TLB_MOD,
        TLB_LOAD,
        TLB_STORE,
        ADDR_ERR_LOAD,
        ADDR_ERR_STORE,
        BUSERR_INSTR,
        BUSERR_DATA,
        SYSCALL,
        BREAKPOINT,
        RES_INSTR,
        CP_UNUSABLE,
        OVERFLOW,
        UTLB_MISS = 32
} ExceptionId;
```

In MIPS terminology, interrupts are a special kind of exception, so exception is the general concept. With interrupts we mean hardware interrupts (e.g. caused by the hardware clock):

```
typedef enum {
        CLOCKINT_0 = 0,
        CLOCKINT_1 = 1,
        CENTRONICS = 3,
        FPGA_CARD = 4,
        DUART = 5,
        SWINT_0 = 6,
```

```
        SWINT_1 = 7
} InterruptId;
```

How are exceptions resp. interrupts processed? Whenever a common exception occurs, the MIPS processor jumps to a predefined address according to the kind of exception (see Figure 10 on page 29). If a Reset happens, it jumps to the address stored in 0xbfc00000. An UTLBMiss exception (a virtual address in user space cannot be translated into a physical address because there is no entry in the translation look-aside buffer (tlb) - see AOSC, section 9.4.2.1) causes a branch to the address specified in address 0x80000000. UTLBMiss exceptions are caught by the `UTLBMissHandler` in TMHalAsm.S. All other exceptions are treated by one handler (`generalExceptionHandler` in TMHalAsm.S) located at address 0x80000080. The steps performed in `generalExceptionHandler` are:

1. Save registers to be modified by the exception handler.
2. Set stack pointer and frame pointer on the exception stack.
3. Save context of the current thread.
4. Look up the exception handler table to get the address of the specific exception handler.
5. Call the specific exception handler.
6. Restore context of the current thread.

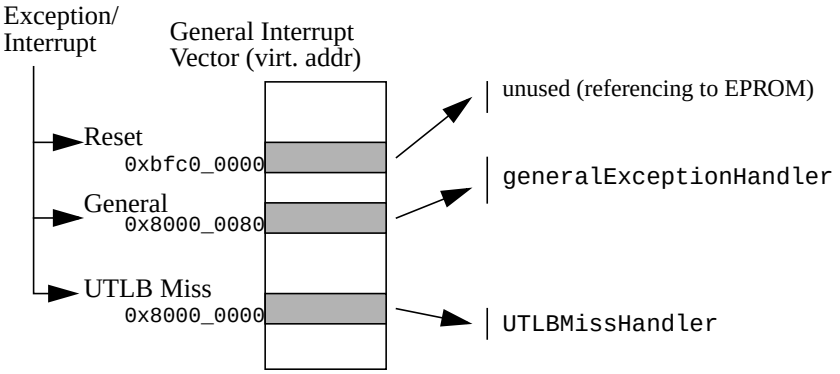


FIGURE 10. Interrupt and exception processing

A specific exception handler can be installed by `tmSetExceptionHandler` in TMHal.c:

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

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

Since hardware interrupts are a special kind of exception a global interrupt handler (`hwExceptionHandler` in TMHalAsm.S) is installed by

```
tmSetExceptionHandler(HARDWARE_INT, hwExceptionHandler);
```

Similar to common exception handling there is a jump table containing the start addresses of specific interrupt handlers. Specific interrupt handlers are installed by

```
typedef void (*InterruptHandler)(void* arg);

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

Two functions must be mentioned concerning the initialization process of Topsy. The function `initBasicExceptions` located in TMInit.c and called by the start-up code writes the correct start addresses of `generalExceptionHandler` resp. `UTLBMissHandler` to the address 0x80000080 resp. 0x80000000. Additionally, for each exception resp. interrupt, a dummy exception resp. interrupt handler is installed. The second function `tmInstallErrorHandlers` (TMError.c) registers two exception handlers: for hardware interrupts (`hwExceptionHandler`) and for the Syscall exception caused by the `syscall` instruction (`syscallExceptionHandler` in TMHalAsm.S). The syscall exception handler is needed to implement `tmMsgSend` and `tmMsgRecv`. Also the clock interrupt handler is set (`tmClockHandler` in TMScheduler.c - invokes only `schedule`). Furthermore, four exception handlers are installed to catch errors and handle faulting threads.

3.8.2 Clock Interface

The 82C54 which is placed on the experimental board 7RS385 we are using is able to generate accurate time delays under software control. A total of six different modes can be programmed for the timer, however, only the rate generator mode is of interest for Topsy. This mode allows the generation of periodic interrupts at constant time intervals.

The programming interface to the hardware clock is given below:

```
/* The 2 timers of the 82C54 programmable timer */
typedef enum { CLOCK0, CLOCK1 } ClockId;

/* Used modes for the 82C54, currently a single mode */
typedef enum { RATEGENERATOR, SWTRIGGER} ClockMode;

/* Configuration of the 82C54 (each counter sep.) */
Error setClockValue(ClockId cId,
                    int period, /* given in [ms] */
                    ClockMode cMode);

/* Reset after a clock interrupt */
void tmResetClockInterrupt(ClockId cId);
```

The clock is configured by a call to `setClockValue` with the timer identifier, the required period in milliseconds, and finally the clock mode. As the 82C54 is driven by an experimental board, it is necessary to ‘reset’ the clock each time an interrupt occurs with a call to tmResetClockInterrupt.

4. MEMORY MANAGEMENT

4.1 Overview

Memory management is one of the central topics when building an operating system. This chapter describes the memory manager implemented in Topsy from different views (see figure 11). The interface to the user, to the other kernel subsystems, and to the hardware is defined. Furthermore, the internal structure of the memory manager (hardware dependent as well as hardware independent parts) is presented.

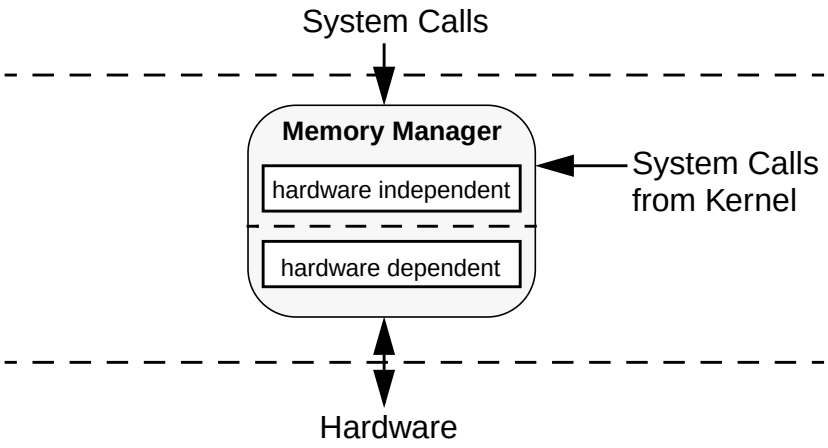


FIGURE 11. Memory management overview

Let us first discuss the common memory model used in Topsy. There is one virtual memory address space which is split up in a user and a kernel address space (virtual memory is explained in depth in chapter 9, AOSC). That means a certain address range is reserved for kernel threads, another region in the virtual address space can be used by user and kernel threads (cf. Figure 12 on page 31).

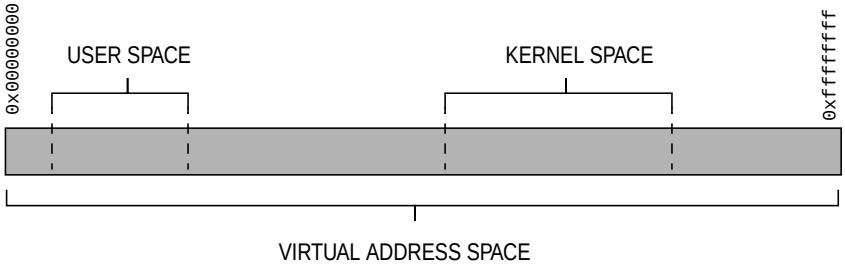


FIGURE 12. User and kernel address space (example)

Virtual memory is broken into fixed-size blocks called *pages* (see AOSC, section 9.4). Physical memory is also divided up in blocks of the same size, which are called *frames*. To map virtual (also called logical) addresses to physical addresses, a page table is necessary. The page table contains the base address of each page in physical memory.

On the MIPS processor in the practical course, we are using a special and fast hardware cache. For this purpose, a translation look-aside buffer (TLB for short - refer to

section 9.4.2.1, AOSC) is realized. The TLB contains 64 entries, each entry consists of two parts: the key (a virtual address) and the value (a physical address).

To understand the mechanism how virtual and physical addresses are connected, let us have a look at Figure 13 on page 32. The picture represents the address mapping supported by the MIPS R3000A processor. The virtual address space consists of four segments: three segments can only be accessed in kernel mode (kseg0, kseg1, kseg2), one is accessible in user and kernel mode (kuseg). The segments kseg0 and kseg1 are directly mapped to the first 512MB in physical memory, that means, e.g., the virtual address 0x80000000 is transformed by the hardware to the physical address 0x00000000. No look-up in the TLB is necessary. The segments kuseg and kseg2, on the other hand, can be mapped anywhere in the physical address space. To translate a virtual address within kuseg or kseg2, the MIPS processor uses the TLB. Obviously, the size of the TLB is restricted and whenever a virtual address has no entry in the TLB, a special exception occurs. The operating system is responsible for updating the TLB with an new entry which refers to the corresponding virtual address. (please refer to section 9.4, AOSC, to get more detailed information about paging).

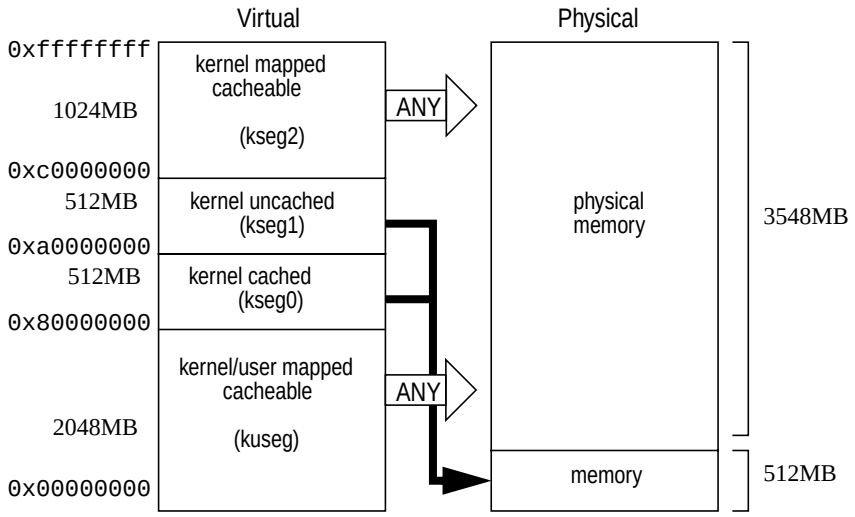


FIGURE 13. Virtual to physical address mapping of the MIPS R3000A processor (*The IDT3051, R3052, RISController Hardware User's Manual*, 1994, Integrated Device Technology, Inc., page 4-4)

4.2 Memory Manager System Calls

The memory manager provides two system calls for user threads (which can also be used in kernel mode): `vmAlloc` and `vmFree`. They are used to allocate and free so-called *virtual memory regions* which are memory blocks in the virtual address space. Each block must have a size that is a multiple of the page size. Note that user threads (as well as kernel threads) are not concerned with virtual address mapping. Whenever a user thread has reserved a memory region in the virtual address space, it can arbitrarily perform read and write operations on this memory block.

4.2.1 vmAlloc

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

This routine reserves a virtual memory region of the desired size (the size is internally rounded up to a multiple of the page size). The start address is returned in the variable specified by the first parameter. If a user thread calls `vmAlloc`, the virtual memory region is allocated in user space, kernel threads reserve regions in the kernel space. In case of an error, `VM_ALLOCFAILED` is returned, otherwise `VM_ALLOCOK` signals the proper execution of `vmAlloc`.

4.2.2 vmFree

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

A previously with `vmAlloc` reserved virtual memory region is deallocated. In the case of an error `VM_FREEFAILED` is returned, otherwise `VM_FREEOK`.

Kernel threads are allowed to deallocate any region while user threads may only free their own regions.

4.3 System Calls (Kernel Mode only)

Three additional system calls operating on virtual memory regions are available for kernel threads: `vmMove`, `vmProtect` and `vmCleanup`. The first routine `vmMove` is needed by the thread manager in `tmStart` to move an allocated stack from kernel to user space. Furthermore, the thread manager uses `vmCleanup` whenever a thread exits or is killed. `vmProtect` is not supported in this version of Topsy.

4.3.1 vmMove

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

With this function a region can be moved from kernel to user space. The start address of the region is given by the value in `*addressPtr`, the destination address space is derived from the `newOwner` parameter (in fact only moves from kernel to user make sense). `vmMove` deletes the old region and reserves a new one of the same size in the desired address space. The last parameter explicitly defines the owner thread of the new region. If the specified region is already located in the destination address space no movement takes place, only the owner changes. The contents of the region is the same after this operation.

The return value is `VM_MOVEFAILED` (an error occured) or `VM_MOVEOK` (no error).

4.3.2 vmProtect

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

A region or parts of a region can be protected in two ways: read only or inaccessible. The first parameter specifies the start address (within a region), the second parameter defines the length of the protected area (internally adapted to a multiple of the page-size); if the size 0 is passed the whole region is protected. For the last argument there a three possible values: `READ_WRITE_REGION` (unprotect), `READ_ONLY_REGION` (write protect) and `PROTECTED_REGION` (inaccessible).

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

The return values are corresponding to the other functions VM_PROTECTFAILED in case of failure or VM_PROTECTOK (on success).

vmProtect is not implemented in the Topsy version used in the practical course!

4.3.3 vmCleanup

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

All former allocated virtual memory regions which are owned by the specified thread are deallocated. On success, VM_CLEANUPOK is returned, in the case of failure VM_CLEANUPFAILED.

4.4 Heap Manager Functions

For kernel use only, two functions are exported by the heap manager, a submodule of the memory manager. The heap manager provides allocating and deallocating memory blocks in byte granularity. To achieve better system performance these functions are not implemented as system calls, but are called directly from kernel threads. Hence, a synchronization mechanism (spinlocks, see appendix) must be used in order to guarantee exclusive access to the internal data structures.

The kernel heap is used throughout all kernel modules for dynamically growing lists and various descriptors.

4.4.1 hmAlloc

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

A piece of internal kernel heap memory is allocated. The second parameter specifies the desired size in bytes. The start address of the allocated area is written to *addressPtr. Return values are HM_ALLOCFAILED and HM_ALLOCOK.

4.4.2 hmFree

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

Deallocates a piece of internal kernel memory reserved by kmAlloc. If any error occurs HM_FREEFAILED is returned, otherwise HM_FREEOK.

4.5 Internal Design

This section deals with the general design of the memory manager. The aim is to give an overview as basis for the following sections where the single parts of the memory subsystem are described in more detail.

4.5.1 Module structure

The internal module structure of the memory manager is shown in figure 14.

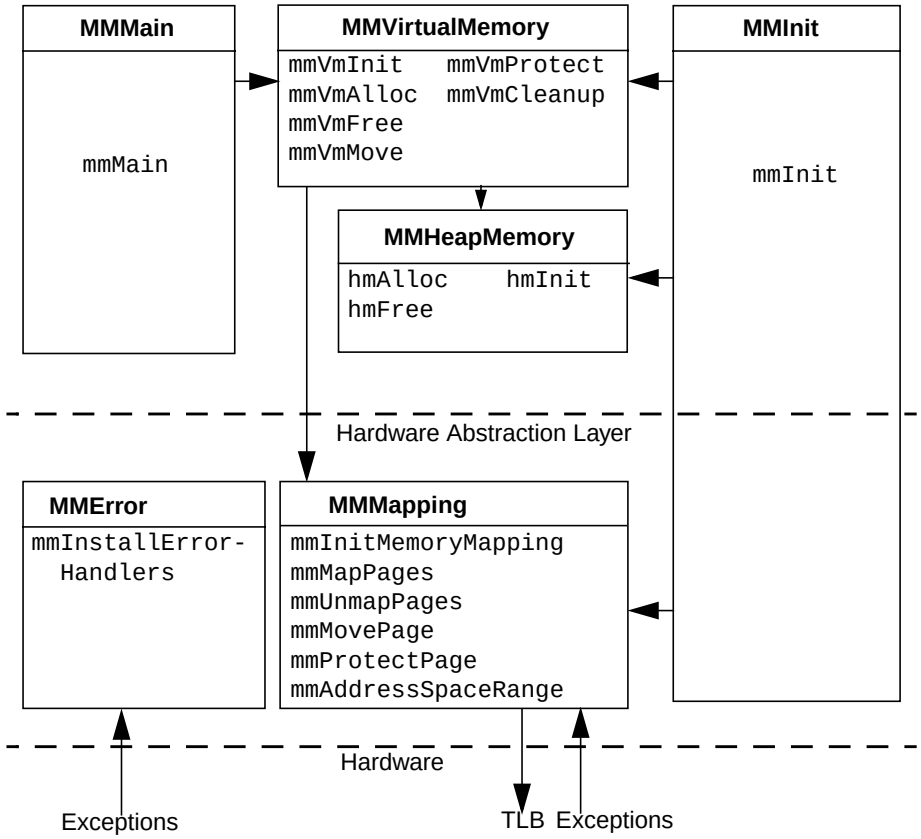


FIGURE 14. Module Structure and Exported Functions of the Submodules of the Memory Manager

Above the hardware abstraction layer there are three main modules: the message dispatcher (module `MMMain`), the management of *virtual memory regions* (module `MMVirtualMemory`) and the management of *kernel heap memory* (module `MMHeapMemory`). The message dispatcher is implemented as a simple endless loop which waits for the next message for the memory manager, decodes it and invokes the corresponding functions in `MMVirtualMemory`. Therefore the exported routine `mmMain` is the main routine of the memory manager (Synopsis: `void mmMain(ThreadArg arg)`), called from the thread manager to start the memory kernel subsystem. But before this can be done all internal data structures must be initialized by the function `mmInit` (for detailed description see section “Initialization Procedure” on page 40). The hardware abstraction layer contains two modules: an error handler (`MMError`) and the memory mapping functions (`MMMapping`). The error handler is responsible to react to exceptions like address error and bus error. In this case the current thread causing the exception is killed. The functionality concerning memory mapping (virtual addresses to physical addresses) is very machine dependent and for this reason placed in the hardware abstraction layer.

4.5.2 Imported Functions

The memory manager needs to communicate with other (user or kernel) threads. Therefore the syscalls `tmMsgSend` and `tmMsgRecv` are used. For exception handling also two other functions are required: `tmSetExceptionHandler` and `kSend`. The first routine installs an exception handler for special exceptions.

kSend is necessary to send a message to the thread manager while handling exceptions. Since in the case of an unrecoverable error (e.g. a bus error) a thread has to be killed, a kill message is sent via kSend to the thread manager (tmKill must not be used because it generates an exception (syscall) itself). All functions or messages are exported by the thread manager.

4.6 Hardware Independent Functionality

In this section the management of the kernel heap memory and of the virtual memory regions is described.

4.6.1 Kernel Heap Memory

The functions exported by the module MMHeapMemory are

```
Error hmInit(Address addr);
Error hmAlloc(Address* addressPtr,
    unsigned long int size);
Error hmFree(Address address);
```

hmAlloc and hmFree are explained in section “Heap Manager Functions” on page 34, hmInit is responsible for the correct initialization of the heap data structure. This data structure is a simple list, shown in Figure 15 on page 36.

Every list entry has a field AreaStatus which contains information about the memory block lying between this list entry and the next list entry. Either the memory block is allocated by a former hmAlloc (HM_ALLOCATED) or is unused (HM_FREED). Having called hmInit the list contains two entries. The first is stored at the beginning of the heap area and the last is placed at the end of the heap. The memory block between them is the heap space that can be allocated by hmAlloc. Therefore, the field AreaStatus of the first list entry is set to HM_FREED.

Allocating a memory block is very simple. First all list entries are scanned until a block of the desired size (or bigger) has been found (section 9.3, AOSC. If this block is greater than required, it is split up into two blocks (an allocated one of the specified size and a free one of the remaining size) by inserting a new list entry.

By invoking hmFree a former reserved memory block is set to HM_FREED. To avoid heap memory fragmentation contiguous free blocks are automatically merged; this procedure is only performed when hmAlloc wasn’t able to find an area not big enough.

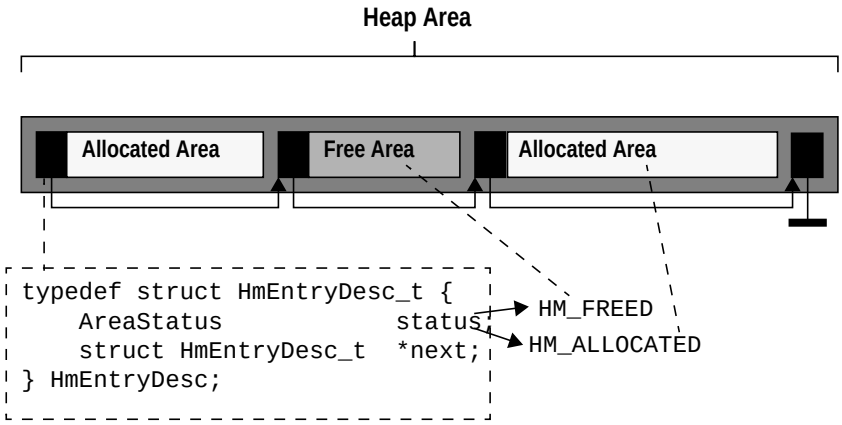


FIGURE 15. Kernel Heap Memory Management

An important issue concerning the differences between kernel heap memory and virtual memory regions has to be mentioned yet. `hmAlloc` and `hmFree` (also `hmInit`) are not implemented as system calls but can directly be called by the kernel threads. Hence, a protection mechanism is necessary when two threads operate on the heap in quasi parallel manner. The heap manager uses spinlocks (see appendix) to solve this problem. Nevertheless, the memory space used by the heap is an ordinary virtual memory region (in kernel space) allocated in `mmVmInit`.

4.6.2 Virtual Memory Regions

Virtual memory regions are managed separately for user and kernel space. For each address space two lists are maintained according to allocated and free virtual memory regions. The lists are implemented by using the abstract data type `List` (see appendix). Each list entry contains the base address of the first page of the region, the size of the region (measured in pages), a protection mode which is modified by `vmProtect,` and the thread identifier of the owner. The relevant type definitions are given below:

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

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

The mechanism of allocating and deallocating virtual memory regions is similar to heap memory. A `mmVmAlloc` call causes the memory manager to search a fitting area in the free list of the corresponding address space. If necessary the found region is split up in an allocated and a free area. Deallocating a virtual memory region is performed by moving the corresponding list entry from the *allocated* list to the *free* list (of the address space). `mmVmInit`, the initialization procedure of the module `MMVirtualMemory` is explained in section 4.8 on page 40.

The routines `mmVmMove`, `mmVmProtect` and `mmVmCleanup` have almost the same functionality as the corresponding system calls. All functions embedded in the module `MMVirtualMemory` are listed below:

```
Address mmVmGetHeapAddress(Address kernelDataStart,
    unsigned long int kernelDataSize);
Error  mmVmInit(Address kernelCodeStart,
    unsigned long int kernelCodeSize,
    Address kernelDataStart,
    unsigned long int kernelDataSize,
    Address userCodeStart,
    unsigned long int userCodeSize,
    Address userDataStart,
    unsigned long int userDataSize,
    Address* mmStack, Address* tmStack);
```

```
Error mmVmAlloc(Address* addressPtr,
    unsigned long int size, ThreadId owner);
Error mmVmFree(Address address,
    ThreadId claimsToBeOwner);
Error mmVmMove(Address* addressPtr, ThreadId newOwner);
Error mmVmProtect(Address startAdr,
    unsigned long int size,
    ProtectionMode pmode, ThreadId owner);
Error mmVmCleanup(ThreadId id);
```

The function `mmVmGetHeapAddress` is important for the initialization process in order to reserve the heap area in the kernel space.

4.7 Hardware Abstraction Layer (HAL)

The hardware abstraction layer of the memory manager is mainly concerned with the mapping of virtual to physical addresses. As mentioned before this could be implemented by using a page table. However, the version of Topsy which is used in the practical course comes with a very simple mechanism to handle address translation. This approach, we call it direct mapping, is explained in section 4.7.2 on page 39. Nevertheless, it should be possible to extend Topsy for supporting demand paging (see AOSC, section 9.2). For this reason and with portability in mind we designed the interface of the hardware abstraction layer very flexible.

To understand the meaning of the functions firstly the interface is described as if paging would be supported and implemented. Afterwards we deal with the concrete implementation of direct mapping and the way the HAL interface was realized.

4.7.1 Interface

The interface of the memory management HAL contains six routines: two are necessary for the initialization procedure, four concern the manipulation of the page table. The functions are declared in the header file MMMapping.h.

The first function to be called in the bootstrap process is `mmInitMemoryMapping`:

```
Error mmInitMemoryMapping(Address  codeAddr,
    unsigned long int  codeSize,
    Address  dataAddr,
    unsigned long int  dataSize,
    Address userLoadedInKernelAt);
```

It initializes the page table data structures, collects all free pages and maps the user program to a fixed virtual address. The first four parameters specify the physical start address of the user code segment, its size, the physical start address of the user data segment and the size of it. The start address of the user program in the kernel space (after loading Topsy on the experimental board) is passed as last parameter.

Important for the initialization of the virtual memory region management (`mmVmInit`) is the following routine:

```
Error mmAddressSpaceRange(AddressSpace space,
    Address*  addressPtr,
    unsigned long int*  sizePtr);
```

With `mmAddressSpaceRange` it can be found out which space in the virtual memory space a special address space covers. This routine shields processor specific information from the *virtual memory regions* module (`MMVirtualMemory`).

To make changes to the page table or to page table entries four functions are provided by the memory manager HAL:

```
typedef enum {READ_WRITE_PAGE, READ_ONLY_PAGE, PROTECTED_PAGE,
    INVALID_PAGE} PageStatus;

Error mmMapPages(Page startPage, Page endPage,
    PageStatus pstat);
Error mmUnmapPages(Page startPage, Page nOfPages);
Error mmMovePage(Page from, Page to);
Error mmProtectPage(Page page, ProtectionMode pmode);
```

The first function (`mmMapPages`) sets the entries in the page table for the virtual pages `startAdr` to `endAdr`. If the pages are not marked as invalid (last argument `INVALID_PAGE`), the virtual pages are mapped to physical pages if necessary. The status of the pages is set to the declared one. Allowed are `READ_WRITE_PAGE`, `READ_ONLY_PAGE`, `PROTECTED_PAGE` and `INVALID_PAGE` (stands for unmapped pages). Since virtual memory regions are dynamic it must be possible to unmap pages. This is done by the function `mmUnmapPages` (frees all pages in the interval `startAdr` to `endAdr`). Finally there is the need to translate pages to another virtual address, that means to link the mapped physical pages of a region with new virtual addresses (compare `vmMove`). Therefore `mmMovePage` is available. It copies the page table entries for the virtual addresses from `startAdr` to `endAdr` to the corresponding page table fields for the addresses `destAdr` to `destAdr+(endAdr-startAdr)`. After copying the entries for the original region are set to invalid. The fourth function `mmProtectPage` is used by `vmProtect`. It sets the status of a page (cf. data type `PageStatus`).

4.7.2 Direct Mapped Memory Management

The term *direct mapping* describes the method of translating virtual to physical addresses which was chosen for the implementation used in the practical course. The address translation is done in a statical or direct fashion which is fixed up at startup time. Recall Figure 13 on page 32. By definition, kernel space is located in the segment kseg0, so the address translation is performed by the MIPS hardware. The simplest way to map user space is to initialize the TLB with a fixed mapping. Since the MIPS TLB contains 64 entries and the page size is 4kB, a virtual memory area up to 256kB can be directly mapped in this manner. And that is what `mmInitMemoryMapping` does: a 256kB, contiguous region at the beginning (address 0x1000) of the segment kuseg is mapped to physical memory by setting up the 64 TLB entries correctly (TLB accesses are implemented in file tlb.S). Hence, no software exception handler is necessary to catch TLB miss exceptions, which are caused by a virtual address not stored in the TLB. The user space contains 256kB by definition which are mapped via TLB. The kernel space is located in kseg0 which is mapped directly by the MIPS hardware and does not use the TLB. The reason to start the user address space at 4k and not at address zero is to catch null pointer dereferencing by a UTLB miss exception.

This approach has implications to the functionality of the HAL interface. `mmMapPages` as well as `mmUnmapPages` consists only of a return statement because the mapping is fixed. `mmMovePage` is realized by copying a memory block. `mmProtectPage`, on the other side, cannot be realized in this approach and therefore always returns `MM_PROTECTPAGEFAILED`. Furthermore, the initialization procedure `mmInitMemoryMapping` sets up the TLB entries and copies the user code and data to the user space (note that the user program is located in kseg0 after having loaded Topsy on the experimental board).

For direct mapped systems, the page size for virtual memory regions can be defined arbitrarily. In order to distinguish between a page size defined in this way and a physically defined page size, we introduce the notion of a logical page size. For direct mapped systems this logical page size may be smaller the the physical page size (if the processor uses pages at all; most microcontrollers do not fragment the memory into page-sized chunks). If demand paging is used, the logical pagesize must be equal or larger than the physical page size.

4.7.3 Memory Layout

To get a deeper understanding of the Topsy implementation figure Figure 16 on page 41 depicts an example for the memory layout while running Topsy (direct mapped memory). The locations and sizes depend on the hardware, kernel constants, code and data size produced by the compiler and the pagesize of the system.

The following constants were used:

```
#define K 1024
#define PAGESIZE 4*K
#define KERNELHEAPSIZE 10*PAGESIZE
```

4.8 Initialization Procedure

The bootstrapping process is mostly a difficult and ugly procedure because several hardware dependencies have to be considered. A problem in this OS design is that the memory manager is an independent thread scheduled by the thread manager. But on the other side, the thread manager needs functions of the memory manager to build his own data structures. For this reason `mmInit` has to be invoked in the start-up code before calling `tmInit`, the initialization procedure of the thread manager. Written as C prototype `mmInit` is defined in the following manner:

```
void mmInit(SegMapPtr segMapPtr, Address* mmStackPtr,
    Address* tmStackPtr);
```

Two values are returned (by reference): the address of the stack for the memory manager (`mmStackPtr`) and the address of the thread manager stack (`tmStackPtr`). The first argument is a pointer to a structure which contains information about the loaded kernel and user program:

```
typedef struct SegMapDescriptor_t {
    unsigned long int  kernelCodeSize;
    Address            kernelCodeStart;
    unsigned long int  kernelDataSize;
    Address            kernelDataStart;
    unsigned long int  userCodeSize;
    Address            userCodeStart;
    unsigned long int  userDataSize;
    Address            userDataStart;
    Address            userJumpAddress;
    Address            userLoadedInKernelAt;
} SegMapDescriptor;
```

The field `userJumpAddress` refers to the start address of the user program (needed by `tmMain` in file TMMain.c). The last field `userLoadedInKernelAt` contains the address in the segment kseg0 where the user program is loaded at.

How `mmInit` works can be described in the following steps:

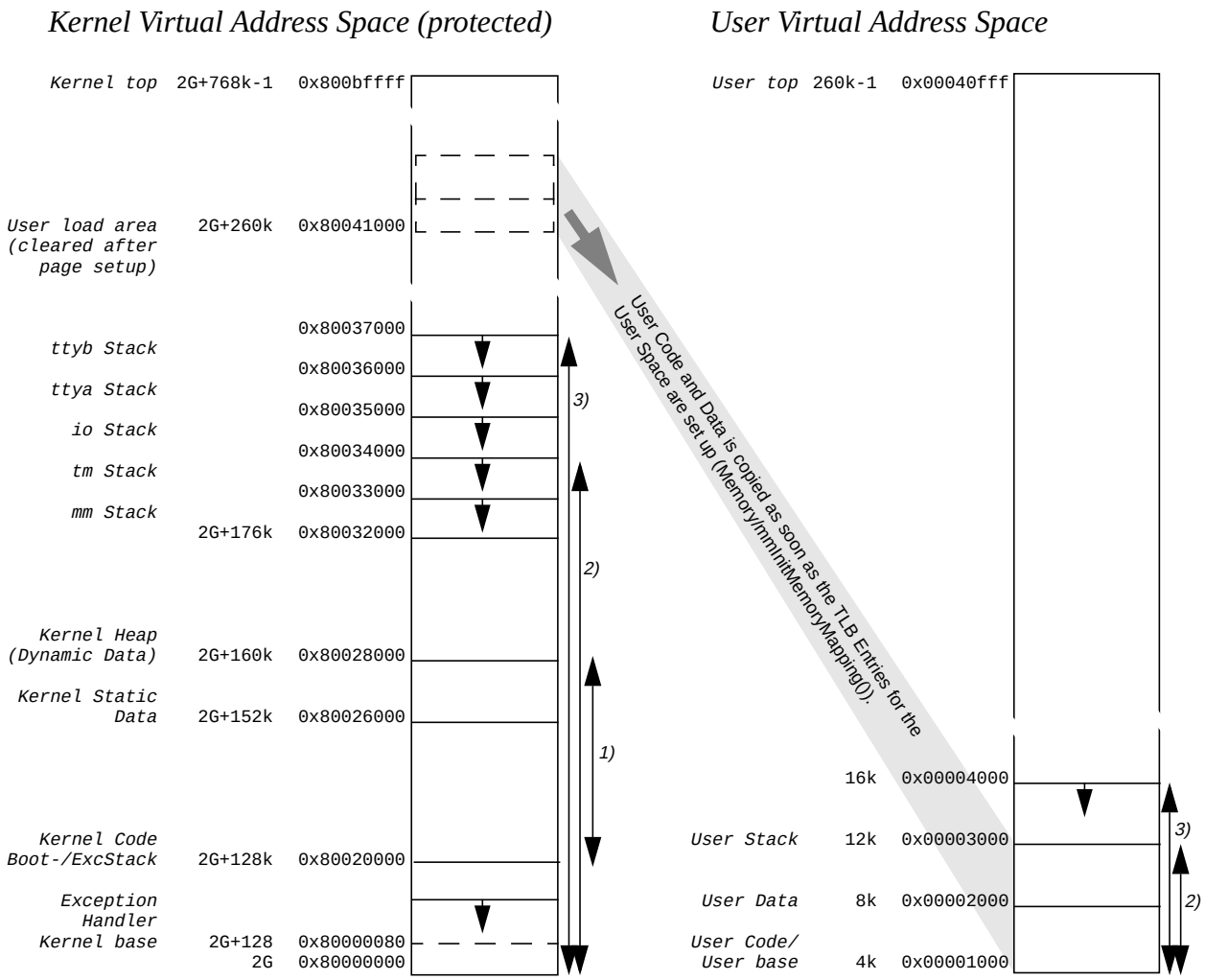


FIGURE 16. Memory layout for the MIPS R3000 using paging to create a direct memory mapping (due to the 64 TLB entries, user memory is limited to 256kB). The numbered arrows mark the initialization of the regions after loading (1), after tmInit() (2), and after full kernel setup (3) (i.e. all kernel threads wait for work).

1. The exception handlers concerning memory management are installed to catch bus, address and TLB errors. This is performed by the function `mmInstall-ErrorHandlers` in file MMError.c.
2. By invoking `mmInitMemoryMapping` (in file MMDirectMapping.c) the 64 TLB entries are set correctly and the user program is transferred to user space. (Remember that kernel code and data as well as user code and data are loaded in the segment kseg0. Kernel code and data remain there, while user code and data have to be moved to the segment kuseg).
3. The kernel heap is built. Firstly, a virtual memory region for the heap is installed by calling `mmVmGetHeapAddress` (file MMVirtualMemory.c) and afterwards `hmInit` (file MMHeapMemory.c) is invoked.
4. The module `MMVirtualMemory` is initialized. In the kernel address space there are initially seven virtual memory regions: the boot/exception stack, kernel code, kernel data, kernel heap, memory manager stack, thread manager stack and

a free region covering the kernel space that is left. The user space contains at the beginning a region for user code, a region for user data and a free region representing the unused user space which can be allocated by `vmAlloc`.

5. I/O Subsystem and Driver Interface

The I/O subsystem is divided into a management part and the actual drivers. These units are running separately as independent threads, i.e. there is one thread (I/O thread) responsible for driver independent functions (`ioOpen` and `ioClose`) and further threads (driver threads) implementing the interface to specific hardware devices (`ioRead`, `ioWrite`, and `ioInit`). User or kernel threads may ask the I/O thread for a certain device to be non-exclusively opened which returns the thread id of the driver thread (if it exists). This thread id allows another thread to communicate with the specific driver.

5.1 System Calls

The following section describes the system calls provided by the I/O subsystem. Note that the system calls `ioOpen` and `ioClose` are internally implemented by exchanging messages between the calling thread and the I/O thread (see Syscall.c for details). `ioRead`, `ioWrite`, and `ioInit`, on the other hand, directly communicate with driver threads.

5.1.1 Open

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

`ioOpen` suspends the caller until the desired device specified by a number is open. Device numbers are defined as constants in IO.h (available are `IO_SERIAL_A`, `IO_SERIAL_B` and `IO_CONSOLE`). If the operation succeeds, `IO_OPENOK` is returned and the id of the specific driver thread is stored in `*id` (return value by reference). Otherwise `IO_OPENFAILED` indicates an error.

5.1.2 Close

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

`ioClose` forwards a close message to the device thread which is responsible for the appropriate action (it may remove pending requests, etc.). Returns `IO_CLOSEOK` or `IO_CLOSEFAILED`.

5.1.3 Read

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

Maximally `*nOfBytes` bytes are read from the device specified by the thread id and written to the buffer whose start address is passed as second parameter. Ensure that the buffer is able to store at least `*nOfBytes` bytes! Since the callee of `ioRead` may not know how many bytes are available, afterwards `*nOfBytes` contains the number of bytes actually read (zero means that there was no character available). On success, `ioRead` returns `IO_READOK`, otherwise `IO_READFAILED`.

5.1.4 Write

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

Writes `*nOfBytes` bytes stored in `buffer` (array of characters) to the specified device. Similar to `ioRead` `*nOfBytes` contains the number of bytes actually written. In the case of failure `IO_WRITEFAILED` is returned, `IO_WRITEOK` indicates the proper execution of `ioWrite`.

5.1.5 Init

```
SyscallError ioInit(ThreadId id);
```

The `ioInit` system call is used to start up and initialize a device.

5.1.6 Control Functions

Other functions (in Unix they are called io-controls, 'ioctl') which may set device parameters or query the status of a device, are defined individually. For example, an UART (universal asynchronous receiver transmitter, serial device) may support a function

```
int available(ThreadId device);
```

which returns the number of bytes that can be read from this device. Since multiple threads may read from the same device, the actual number of bytes read may be smaller than reported by `available`. The mapping of these functions to messages and the corresponding decoding must be implemented in libraries (like Syscall.c) and in the `handleMsg` function of the driver.

5.2 I/O Subsystem Design

As mentioned before, the I/O subsystem consists of a common management part and device specific parts (see Figure 17 on page 44). The module `IOMain` represents the code for the I/O thread. Module `IODevice` serves as a basis for the driver threads; it contains the main routine for *all* driver threads. The specific driver code is separated from that (e.g. SCN2681_DUART.c). Additionally, two modules exist (`IOConsole` and `IOHal`) which provide simple stand-alone output routines. This is needed to print error, warning, info messages, etc.

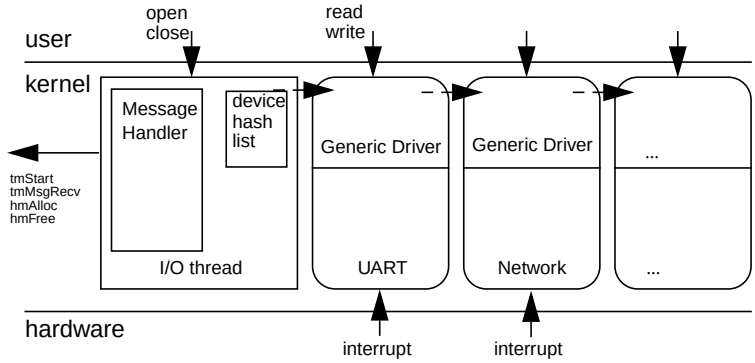


FIGURE 17. I/O thread and its list of driver threads

5.2.1 IOMain

The main procedure of the I/O thread looks like the ones of the memory manager and the thread manager:

```
void ioMain(ThreadArg arg);
```

Firstly, this routine starts the driver threads which are specified in the array `ioDeviceTable`. Each device is described by a structure of type `IODeviceDesc` which is explained in the next subsection. Also the interrupt handlers of the specific drivers are installed. The thread ids of the driver threads are stored in a hash table (see appendix). Afterwards, `ioMain` executes an endless loop expecting `ioOpen` and `ioClose` messages. Receiving an `ioOpen` message causes the I/O thread to look up the hash table in order to find the thread id of the driver thread related to the

given device number. Closing a device is accomplished by forwarding the close message to the specific driver thread.

5.2.2 IODevice

The underlying concept of the driver threads is generic, i.e. `IODevice` makes a generic driver available, in an object oriented manner (although C is not an object oriented programming language). IODevice.c exports the function `ioDeviceMain`:

```
void ioDeviceMain(IODevice this);
```

`ioMain` starts and initializes all driver threads via `tmStart`, and `ioDeviceMain` is specified as start function. However, the functionality `ioDeviceMain` provides is very simple. It handles the driver specific messages (`ioRead`, `ioWrite`, `ioInit` - also close and special messages are managed) und jumps to the corresponding driver subroutines; it also ensures that user threads don't read or write kernel memory. At this point the question arises how `ioDeviceMain` gets the information where the specific routines are located. The answer is that a parameter of type `IODevice` is passed to `ioDeviceMain` when the driver thread is created. `IODevice` is defined as a pointer to the following structure:

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

For each device such a structure exists, stored and specified in the array `ioDeviceTable` in IOMain.c. Hence, a device is defined by its name (`name`), a base address for memory-mapped I/O (see AOSC, section 12.2), a buffer where data can be stored, the interrupt number (`interrupt`) for which the specific interrupt handler (`interruptHandler` - a function pointer) may be installed, and the specific routines to execute read, write, close and init requests (`read`, `write`, `close`, `init` - also function pointers). Additionally, a pointer to a function can be specified (`handleMsg`) which handles device dependent messages (for example to implement the `available` function mentioned before). Furthermore, the field `extension` in the `IODeviceDesc` structure can be used to store additional data (optional).

If an interrupt number and an interrupt handler (this is an ordinary C function) are specified in `IODevice`, the handler's address and a pointer to the IODevice structure are stored in the interrupt table (TMHal.c). Whenever the interrupt is activated, the pointer to the structure is passed as an argument to the interrupt-handler in order to avoid global access to buffers and other private resources of a driver.

Using this generic driver concept, it is possible to support devices with multiple instances like terminals or disks. On initialization, only the base address for memory-mapped I/O, the name and the device number need to be specified for a single instance. As an example, the IDT/MIPS board supports two serial lines that are operated by the same driver code.

So, what has to be done when writing your own device driver? A new device number has to be created in IO.h (e.g. '`#define MY_DEVICE 2`'), the constant `IO_DEVCOUNT` must be incremented (IO.h) and a new `IODeviceDesc` structure has to be added to the `ioDeviceTable`. At least the functions for reading and writing have to be implemented (and filled in), a message handler is optional. Also an interrupt handler may be supplied or set to `NULL` (see appendix) if it should be ignored. Last but not least the fields `name`, `base` (defines the base address for memory-mapped I/O) and `interrupt` (if used) have to be specified.

5.2.3 IOConsole

For bootstrapping and debugging purposes, stand-alone output routines are available (i.e. for kernel messages when the I/O subsystem is not running yet):

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

5.2.4 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) for the I/O subsystem needs only to install C-functions as interrupt handlers. This is done using the `tmSetInterruptHandler` HAL-function in TMHal.c.

Processors featuring *I/O Instructions* (e.g. i386) only allow their use in kernel mode. For convenience, the HAL should define functions (macros) to use them in the C language. Most architectures like the MIPS, however, use *memory-mapped I/O* and do not need special instructions.

In IOHal.c, two low level routines (`ioConsoleInit` and `ioConsolePutChar`) can be found which are used by the functions provided by the module `IOConsole`.

5.3 MIPS Drivers

This section describes the drivers (at present there is only one) implemented for the experimental board.

5.3.1 MIPS: Serial Port (UART)

The serial port driver receives read and write messages from other threads. Write requests are written to the device while read requests are satisfied from an input buffer. Incoming characters generate an interrupt which copies the character into the input buffer. The serial line driver uses its buffer in a bounded and circular fashion. Only n-1 bytes of a n- byte buffer can be used in order to avoid race conditions.

The following functions are exported by SCN2681_DUART.c:

```
void devSCN2681_interruptHandler(IODevice this);
Error devSCN2681_init(IODevice this);
Error devSCN2681_read(IODevice this, ThreadId threadId,
    char* buffer, long int* size);
```

```
Error devSCN2681_write(IODevice this, ThreadId threadId,
    char* buffer, long int* size);
Error devSCN2681_close(IODevice this);
```

The interrupt handler reveives incoming characters and writes them to the buffer referenced by the variable `buffer` within the `IODeviceDesc` structure. Note that for the two used ports (ttya and ttyb) exactly the same interrupt is generated. Therefore the interrupt handler has to decide which port is meant. The init procedure configures the serial port and allocates memory for the buffer. Reading and writing are simple buffer operations, and the close routine only returns `IO_CLOSEOK` because no further operations to abort a connection are necessary.

5.4 Other Drivers

5.4.1 Loopback

A loopback driver is provided for testing purposes. For example, network protocols can be tested with this software-only driver that echos what is written into its buffer.

5.4.2 Network Interface Card

A driver for an FPGA-based network interface card is included in the source. For more information on this proprietary hardware device, contact ETH Zürich.

5.4.3 Timer

The framework for a timer driver based on the intel 8254 chip is included. Since this is part of the exercises at ETH Zürich, the distributed code is *not* complete and *not* working!

6. BOOTSTRAPPING TOPSY

This chapter describes the process of loading, initializing and executing the Topsy kernel from hardware power-on to full operation. A few details are MIPS-specific, the general process however applies to most hardware.

The following steps are performed in order to make Topsy runnable on the experimental board:

1. Compilation of the kernel source code and the user program. Note that this is done on the SUN workstation by using a cross compiler (i.e., the C source code is translated into MIPS machine code).
2. Linking of kernel and user object code. The generated kernel file has a special format which is understood by the loader on the experimental board.
3. Additional information concerning the addresses and sizes of code and data segments is appended to the kernel file. This is done by the bootlinker.
4. The extended kernel file is loaded via serial interface on the experimental board. For that there is a special loader code in the EPROM on the MIPS/IDT board.
5. The loader on the experimental board directly jumps to the startup code of Topsy.
6. After the initialization of Topsy, the user program (first user thread) is started.

6.1 Power-on

Turning on a computer system or setting the reset line of the processor to low causes the machine to initialize the *program counter (PC)* with a predefined address. Usually, this address points to a location where non-volatile memory is located (ROM, EPROM, Flash, NV-RAM). On the MIPS R3000, this address (which is in fact a virtual memory address) is translated to the location of an EPROM which contains code to load the OS from an external device (serial port, network, disk).

6.2 Loader

The loader code (located in the EPROM) understands how to access an external device to read the operating system into memory. Obviously, the loader expects the OS program code and data in a predefined format. There are two alternatives: the OS kernel is either in an ‘image-format’ which means that it will be loaded as a contiguous piece into memory or another approach is to add format information to the kernel which will be interpreted by the loader. The later method was chosen by the loader on the MIPS/IDT board 7RS385 we use in the practical course. The format used is called Motorola S-record format. Code and data are represented as a sequence of special records which contain the relevant information. For 32-bit systems, two basic record formats are understood:

- *S3* Data record with 32 bit load address and up to 255 bytes code or data
- *S7* A termination record with 32 bit transfer address

This simple scheme is sufficient to load code and data *anywhere* into memory and jump to a starting point after loading. Important at this point is the fact that Topsy has to be transformed in S-record format (after compiling and linking) in order to load it on the experimental board.

6.3 Linker

When the Topsy OS is compiled (in our case with gcc) from source code to machine code and data, the addressing is done in a relative way for each object file. Once all modules are compiled they must be linked together and the relative addressing is replaced by a known scheme. The linker (gnu-ld) is responsible for this task (address-fixup). It reads a linker script (a simple textfile link.scr) which defines the

base address of the code segment and places the initialized data at the end of the code segment, aligned to the system's pagesize.

Since we distinguish the Topsy OS which runs in kernel mode from the user programs running in user mode we must fixup the user code and data with separate addresses and load it into memory (i.e. user memory is inexistent at load time).

6.4 Bootlinker

The last problem to solve is how to inform the kernel about this memory layout. Because the start and end points of the segments are known only after compiling the kernel, this information must either be patched into the data segment of the kernel or, more elegantly, be written into a separate segment (a meta segment) at a fixed address known by the kernel code. To produce this segment map a tool was created which reads the information provided by the gnu-size command applied to kernel and user programs and writes a new segment with the size and start address of the kernel code, kernel data, user code and user data segments. Additionally, the start address of the user program is appended to the segment map. The start address of this meta segment is of course the same as was compiled into the kernel.

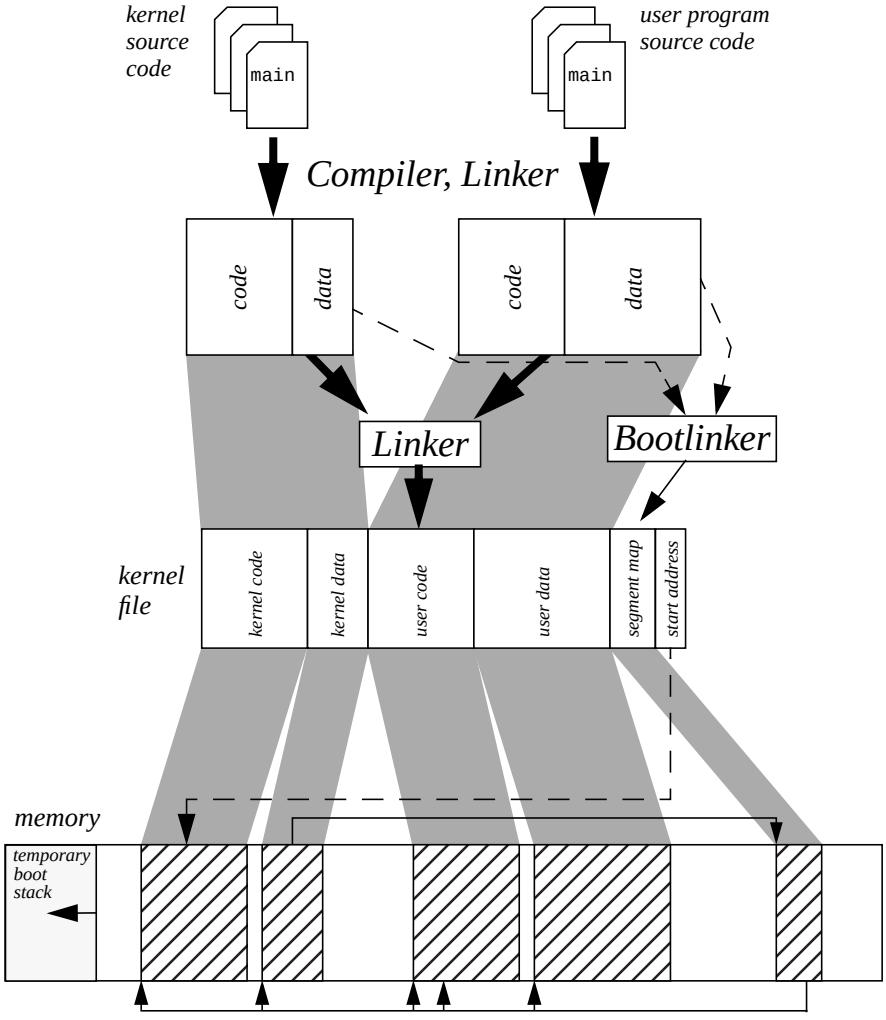


FIGURE 18. Generating a bootable kernel from the source code using compiler and linker

6.5 Start-up Code

The code at the transfer address (S7 record) is a small assembler routine (`__start` in file start.S) that is responsible for setting up a temporary boot stack (which will become later the exception stack) and calling the main function `main` of Topsy. The code for `main` (in file Startup.c) is very simple and presented in Figure 19, "Topsy Main Function," on page 51. Successively, the memory manager and the thread manager are initialized after setting basic exception handlers. Recall that `tmInit` directly yields the control to the first kernel thread. Hence this function doesn't return.

```
void main() {
    Address mmStack, tmStack;
    Address userJump =
        ((SegMapPtr)SEGMAP)->userJumpAddress;

    initBasicExceptions();                /* catch traps early */
    ioConsoleInit();                      /* set baud 9600 etc. */
    ioConsolePutString(BOOTMESSAGE);
    mmInit((SegMapPtr)SEGMAP, &mmStack, &tmStack);/* startup memory init */
    tmInit(mmStack, tmStack, userJump);    /* startup thread init */
    PANIC("init returned");               /* mm/tmInit never returns */
}
```

FIGURE 19. Topsy Main Function

6.6 Loading User Threads

In order to load user threads, the user program is loaded also into kernel space and copied once the virtual memory system of the memory manager is running into user space.

On MIPS systems the user space is located in the lower half of the 32-bit address space (lower 2 GByte). For more details about memory layout see the section "Memory Layout" on page 40.

7. User Programs

7.1 Shell

Topsy comes with a small command line shell enabling the user to start threads, kill threads and get information about threads. The commands available at present are `start`, `exit`, `ps` and `kill`.

7.1.1 start

A new thread is created. The function where the thread execution should start is specified by a name registered in the data structure `userCommands` in file shell.c:

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

static char* argArray[MAXNBOFARGUMENTS+1];

ShellFunction userCommands[NUSERCOMMANDS] = {
    {“shell”, main, (ThreadArg)argArray},
    {“crashme”, crashMeMaster, (ThreadArg)argArray}
};
```

7.1.2 Add a user defined program

In this example two functions are declared. By typing ‘`start shell`’ a new shell thread is created (the shell you are using yet does exist further on and waits for the user program to terminate) By typing ‘`start crashme`’ the Crashme program is started. When writing your own program, you have to add entries to the `userCommands` array. Imagine your program is named `myFunction`. Then the modified array could look like

```
ShellFunction userCommands[NUSERCOMMANDS] = {
    {“shell”, main, (ThreadArg)argArray},
    {“crashme”, crashMeMaster, (ThreadArg)argArray},
    {“myfunc”, myFunction, (ThreadArg)0}
};
```

The first parameter of the new entry specifies the name, the second corresponds to the program to be called and the third parameter defines the argument passed to the

thread to be created. E.g., the command ‘`start myfunc`’ would start a new thread which executes the function `myFunction` without arguments `(ThreadArg)0`.

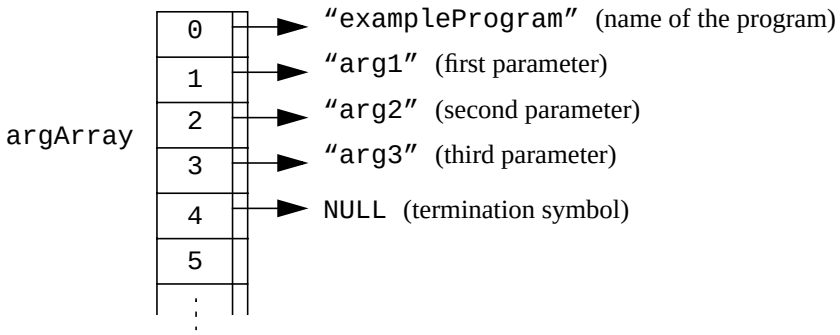


FIGURE 20. Shell Argument Passing Mechanism

Some readers may wonder why `argArray` is passed as argument to the shell function (`main` in file shell.c) and the crashme function (`crashMeMaster`). This is related to an additional feature of `start`. It is possible to enter several parameters in the command line (e.g. ‘`start crashme random`’). Per default the shell parses the arguments and writes them to `argArray` which is defined as a fixed size array of `char` pointers. An example of the layout of `argArray` is given in Figure 20, “Shell Argument Passing Mechanism,” on page 53. The first unused array cell contains `NULL` to indicate the end of the argument list. By specifying `argArray` as parameter in the `userCommands` array a thread is able to take over more than one argument.

7.1.3 start <program> &
(start in the background)

Starts any shell program in the background (e.g. ‘start testprogram &’ would start the thread testprogram in the background). After a & the Shell does not wait for a program to end and can process further commands. If the function has any arguments & causes a setup thread to run in the Shell. This setup thread copies the threads arguments into its stack to memorize them before starting the background shell function.

7.1.4 exit

Exits the current shell thread.

7.1.5 ps

Reports the current process status. No arguments are needed. If no shell function runs it returns the following result:

```
 3  ready       testprogram
-8  blocked     loopback
-7  blocked     fpga
-6  blocked     ttyb
-5  blocked     ttya
-4  ready       idleThread
 2  ready       shell
-3  blocked     ioThread
-2  running     tmThread
-1  blocked     mmThread
```

The first column reports the thread’s identifier, the second the thread’s state (ready, running, or blocked) and the third is its name.

7.1.6 kill

Kills any thread specified by its identifier (e.g. `'kill 3'` kills the testprogram). If any user thread tries to kill a kernel thread it is punished by being killed.

7.1.7 exit

Exits the current shell thread.

7.1.8 help

Prints a help text listing the commands understood by the shell.

7.2 Crashme

Crashme is a remake of the famous Unix test program that brought down almost every commercial system until the developers had a close look at this nifty tool. Like the Unix version, Crashme for Topsy sends *random messages* (garbage) to random threads (even inexisting threads) to exercise the stability of the system. In addition it stresses the memory and thread manager by *allocating memory* and *starting threads* until it gets errors back or the system crashes (hence the name).

8. Extensions of the Topsy OS

8.1 Paged Memory

In the hardware abstraction layer of the memory manager paging on demand could be realized by using page tables. A fast implementation for the MIPS processor is an layered approach with two page tables leading to nested exceptions.

8.2 File Systems

Any file system (AOSC, chapter 11) could run as a user thread processing requests from other user threads. The FS server itself sends/receives messages to/from a storage device driver (disc, network).

8.3 Networking

Protocol stacks may be also implemented as user threads. For performance reasons, zero-copy capabilities should be exploited.

8.4 Real-Time Support

Like there is a simple round-robin scheduler for cooperative and preemptive mode, a version supporting real-time properties could be provided using the interface described in “Scheduler Interface” on page 24. (See AOSC, section 6.1 for real-time scheduling).

A. Common Data Types

Here we list the global data types defined in Topsy.h. These data types are not specific to one particular module but are used throughout all modules.

1.1 Boolean

The type `Boolean` defines two boolean values, which are `TRUE` and `FALSE`.

1.2 Address

The length of an address is processor dependent. On a 16 bit processor, for example, an address has 2 bytes, while on a 32 bit processor it has 4 bytes. In order to keep the source code portable, `Address` is defined as a pointer to the type `void`, i.e. a pointer to anything. Since a pointer exactly corresponds to an address, the definition is machine independent.

1.3 Error

The type `Error` is often used as return value of functions. It characterizes different kinds of errors.

1.4 Register

The same arguments we stated for the definition of `Address` can be applied here. Since the size of registers is processor specific, the type `Register` is introduced to guarantee portability.

1.5 ThreadId

The type `ThreadId` is needed to identify threads. Every thread has an id in order to communicate with other threads. E.g. the addressee of a message is specified by the thread id.

1.6 ThreadArg

When creating a new thread, the parent thread passes an argument (of type `ThreadArg`) to the child thread. In order to be as flexible as possible, `ThreadArg` is defined as pointer to `void`. Hence pointers to any data type (converted by a type cast) can be passed (e.g. `(ThreadArg)pointerToAnything`).

If no argument is expected by the child thread, one can specify the constant `NULL` (a pointer referencing the address zero). `NULL` is always an invalid address, so its often used as pointer return value to indicate an error or something else.

1.7 ThreadMainFunction

Having invoked the system call `tmStart`, the thread manager must know at which point in the user program the new thread is to be started. Therefore `tmStart` is given a start address as parameter. This parameter is of type `ThreadMainFunction` which is a pointer to a function having one argument of type `ThreadArg` and no return value.

Example:

```
void myThreadStartFunction(ThreadArg arg)
{
    ...
} /* void myThreadStartFunction */

void main(void)
{
    ThreadId id;
    ThreadMainFunction functionPtr;
```

```
        functionPtr = myThreadStartFunction;
        ...
        tmStart(&id, functionPtr, (ThreadArg)0, “myThread”);
        ...
    } /* main */
```

After the assignment in the main function, the variable `functionPtr` contains the start address of the function `myThreadStartFunction` - it is used later on to start a thread.

Note that this example serves as illustration how to use function pointers in C; normally, the variable `functionPtr` is unnecessary because the name of the function `myThreadStartFunction` can directly be put into the function call to `tmStart`:

```
tmStart(&id, myThreadStartFunction, (ThreadArg)0,
    “myThread”);
```

B. LIBRARY FUNCTIONS

Topsy has a library of various functions used by the main kernel modules. This library (implemented as module Topsy) embraces simple functions like copying memory blocks as well as complex data structures like lists and hash lists. In the following sections we give an overview over the different functions (sorted by modules).

2.1 Error

The module Error exports four functions:

```
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);
```

They are used to inform the user of the system state and print a message to the console. According to different situations the kind of output varies. E.g. the output

```
*** WARNING [ myFunction, User/myProgram.c:59 ] warning message
```

means that the call to `warning` in the file `User/myProgram.c` at line 59 was executed. The first string `myFunction` refers to the name of the thread where the warning occured. An additional output '`(*in exc*)`' indicates that the error occured while handling an exception caused by the specified thread or by a hardware interrupt, for example

```
*** ERROR [ userThread (*in exc*), Memory/MMError.c:96 ] user
tlb error
```

2.2 HashList

Hashing is an efficient method for looking up data entities characterized by a certain key. The thread manager, e.g., stores information about all threads identified by a special id. Very often there is need to find the information according to one thread. By using a hash list this search process takes very little time. Certainly, a faster implementation can be achieved by using an array for storing thread information. But note that an array is static, i.e. it has a fixed size which cannot be changed while running the program. Hash lists, however, can be implemented in a dynamic fashion, so there is no restriction to the maximum number of data entries.

The module HashList makes hash lists available an abstract data type (ADT). An ADT is characterized by the encapsulation of the internal data structure. Only operations on this data structure are exported. Hence implementation details are hidden from other modules which contributes to abstraction and readability of the program code.

Four operations on hash lists are supported:

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

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

The first operation creates a new hash list with no elements. The second adds an element to an existing hash list. Data is specified by a pointer (has to be type cast!) and its key (given as integer) which serves as search criterion when looking for an special element (`hashListGet`). Having called `hashListGet` the pointer to the data corresponding to the key is returned (exactly: a pointer to this pointer). The function `hashListRemove` deletes an element out of a hash list. This only concerns the key while the data (referenced by the pointer) is not affected by this operation.

2.3 Lock

Locks are a simple synchronization mechanism when using shared memory. Consider a data structure used by two threads. If one thread writes to the data structure while the other thread tries to write too, inconsistencies may occur. To avoid this situation the two threads have to find an agreement who uses the data structure first. By using locks this synchronization problem can be solved.

Spinlocks are a special type of semaphore (see AOSC, section 7.5): integer variables with exactly two allowed values (0 and 1). They also can be seen as boolean variables. In Topsy there are four functions available to handle spinlocks:

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

One to initialize a lock variable (`lockInit`) and two functions for setting (`lock`) and unsetting locks (`unlock`). Note that setting a lock can yield to busy waiting if the lock is already set. The fourth function `lockTry` has the same meaning like `lock` but doesn't wait in the case of a lock already set. It returns `FALSE` if the lock variable has been set, `TRUE` otherwise (success).

2.4 List

Lists are similar to hash lists implemented as abstract data types. A list is an ordered sequence of data elements. One can scan list, one can add an element at a certain position or one can remove an element. Important is the ordering of the list, so there is always a first and a last element (if the list is not empty).

The lists implemented in Topsy have additional features leading to faster list operations. Valid operations are:

```
List listNew();
Error listFree(List list);
Error listAddInFront(List list, void* item,
    Address* hint);
Error listAddAtEnd(List list, void* item,
    Address* hint);
Error listRemove(List list, void* item, Address hint);
Error listMoveToEnd(List list, void* item,
    Address hint);
Error listSwap(List listFrom, List listTo, void* item,
    Address hint);
Error listGetFirst(List list, void** itemPtr);
```

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

Creating a new list resp. deleting an existing list is done via `listNew` resp. `listFree`. Adding an element to a list can be achieved in two ways. `listAddInFront` creates a new element at the head of a list, the counterpart `listAddAtEnd` appends a new element at the end of a list. The data corresponding to an element is given by a pointer to an arbitrary data structure (`void *`). The third parameter `hint` (return parameter by reference) has a special meaning according to efficient list usage. It refers directly to the position of the new element within the list. Thus, removing an element (`listRemove`) can be executed faster by passing `hint` (setting the third parameter to `NULL` signals `listRemove` not to use the '`hint`'). Further operations are `listMoveToEnd` and `listSwap`. The first function moves the element specified by `item` (and referenced by `hint`) to the end of the list. `listSwap` deletes the element specified by `item` (and referenced by `hint`) from list `listFrom` and adds it to the list `listTo`. (Note: the `hint` parameter is only used for reasons of speed up!).

In order to scan a list element by element `listGetFirst` and `listGetNext` are available. `listGetFirst` returns a pointer to the data of the first list element (return value is passed by reference, `itemPtr` is a pointer to a pointer). After execution of `listGetFirst` one is able to get the following elements. `listGetNext` assumes that `*itemPtr` points to the data of the actual element. Afterwards `*itemPtr` contains the data pointer of the next element (if there is none `*itemptr` is filled with `NULL`).

2.5 Support

Support is a collection of simple routines needed in different modules:

```
void byteCopy( Address targetAddress,
    Address sourceAddress, unsigned long int nbBytes);
void zeroOut(Address target, unsigned long int size);
void stringCopy( char* target, char* source );
void stringNCopy( char* target, char* source,
    unsigned long int size);
Boolean testAndSet(Boolean* lockvar);
```

Copying a certain number of bytes form one memory location to another is done by `byteCopy`. `zeroOut` fills a memory region with zeros, `stringCopy` and `stringNCopy` are self explaining (`stringNCopy` copies at most `size-1` characters).

The function `testAndSet` is used to implement locks. In an atomic action it tries to set a boolean variable (referenced `lockVar`) to `TRUE`. If the variable is yet assigned to `TRUE`, `testAndSet` returns `FALSE` (`TRUE` otherwise).

2.6 Syscall

The system calls have been explained in the former chapters. To give a complete definition of the interface we list all system calls (also kernel syscalls which are marked by the comment '`/*kernel*/`') below.

```
SyscallError  vmAlloc(Address *addressPtr,
    unsigned long int  size);
SyscallError  vmFree(Address address);
SyscallError  vmMove(Address *addressPtr,  /*kernel*/
    ThreadId newOwner);
SyscallError  vmProtect(Address  startAdr, /*kernel*/
```

```
    unsigned long int  size, ProtectionMode  pmode);
SyscallError  vmCleanup(ThreadId threadId); /*kernel*/

SyscallError tmMsgSend(ThreadId to, Message *msg);
SyscallError tmMsgRecv(ThreadId* from, MessageId msgId,
    Message* msg, int timeOut);
SyscallError tmStart(ThreadId* id,
    ThreadMainFunction mainFunction,
    ThreadArg parameter, char *name);
SyscallError tmKill(ThreadId id);
void tmExit();
void tmYield();
SyscallError tmGetInfo(ThreadId about, ThreadInfo info,
    Message* reply);

SyscallError ioOpen(int deviceNumber, ThreadId* id);
SyscallError ioClose(ThreadId id);
SyscallError ioRead(ThreadId id, char* buffer,
    unsigned long int* nOfBytes);
SyscallError ioWrite(ThreadId id, char* buffer,
    unsigned long int* nOfBytes);
SyscallError ioInit(ThreadId id);
```