Course of Study: Computer Science

Examiner: Prof. Dr. rer. nat. Klaus Lagally
Supervisor: Dr. rer. nat. Thomas Schöbel-Theuer

Thesis No. 2205

Development of a Process Model for Athomux

Florian Niebling <flo@uue.org>

Institute of Formal Methods in Computer Science (FMI)
Operating Systems Group
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Commenced: 05. April 2004
Completed: 04. November 2004

CR-Classification: D.4.0, D.2.1

Contents

List of Figures

Chapter 1

Introduction

Thomas Schöbel–Theuer has proposed a new design and programming method for operating system kernels [1, 2]. He has developed a lightweight component architecture built on only two universal abstractions called *nests* and *bricks*.

This thesis describes the design and development of the operating system prototype “Athomux” based upon these abstractions.

1.1 Motivation and Objective

The goal of this prototype is to be able to run standard UNIX utility programs such as *grep*, *sed*, a shell like *sh*, a simple text editor and the GNU compiler collection.

Therefore, conventional operating system abstractions such as files, directories and processes have to be emulated using the brick and nest abstractions. To be able to use the functionality provided by the operating system prototype, a subset of the *Portable Operating System Interface* (POSIX) has to be implemented.

In operating systems implementation, the amount of work that needs to be invested into development of device drivers easily exceeds the work on core components such as the filesystem and the process model. To keep the implementation of the Athomux prototype simple, execution as a standalone operating system is not required. By using *GNU/Linux* as a host operating system, it is possible to concentrate on the design and implementation of operating system core components that handle processes and files.

There are two major objectives of this work.

- The design and development of software components based on the bricks and nests architecture that provide traditional operating system abstractions.
- The development of means which allow the execution of the operating system prototype and its user processes in Linux processes.

1.2 Structure of this Work

The thesis is organized as follows.

- **Chapter 2** is divided into two parts. In the first one, conventional operating system abstractions such as files and processes are described. The second part introduces the foundations of instance orientation and the component architecture of the Athomux operating system.
- **Chapter 3** describes how traditional operating system abstractions can be modeled by applying methods of instance orientation. Components are developed which provide abstractions and the functionality to create a POSIX compatible interface for user processes.
- **Chapter 4** covers concepts and solutions of the implementation of the operating system prototype in a Linux environment. These include the mapping and execution of Athomux address spaces containing user programs in Linux processes and the development of a system call interface for the Athomux prototype.
- **Chapter 5** concludes the work with a summary and propositions for future development of the Athomux operating system.

Chapter 2

Prerequisites

This chapter is divided into two parts. First, some abstractions used in traditional UNIX–like operating systems are described. The intention of this part is not to provide a complete survey about operating system techniques in general, but rather to give an overview of functionality that needs to be implemented in the Athomux prototype, with focus on the filesystem and user processes.

The second part introduces the basic principles of instance orientation and the component architecture of the Athomux operating system.

2.1 POSIX Operating System Abstractions

Understanding the creation and meaning of POSIX requires a short review of the evolution of the UNIX operating system[1].

UNIX was developed as a time–sharing operating system at the AT&T Bell Laboratories in the late 1960s [3]. Because of the open architecture of UNIX which included the source code and allowed for extensions of the system, many universities adopted UNIX and added additional functionality to satisfy their needs. In the 1980s, multiple variants of the operating system existed. AT&T tried to end the diversity by combining various versions developed at other universities and corporations into *UNIX System V*, turning it into a commercial product. Since AT&T did not release the *UNIX System V* source code, the University of California (UCB) continued to develop their own UNIX variant, *BSD*.

In the mid-1980s, the danger of incompatibility between the various UNIX variants became apparent as it became increasingly harder to develop applications that were able to run on the various UNIX systems.

As a consequence, the *Portable Operating System Interface* (POSIX) was designed to allow the creation of platform independent applications. POSIX successfully merged the interfaces of the different UNIX variants. Today, almost

[1] Omitting several, certainly interesting chapters of the UNIX evolution that are not vital to emphasise the reasons behind the creation of POSIX.

every operating system has to provide a POSIX interface in order to support the vast number of applications that were written according to the POSIX standard definitions.

Although POSIX has no direct influence on how the operating system itself is designed[2], a POSIX compatible system has to provide certain abstractions to its user programs. In the following paragraphs, the important abstractions that a POSIX compatible operating system needs to provide are introduced.

2.1.1 Processes and Threads

Processes consist of a segmented address space shared by one or more threads of control. A process contains a user program that is currently being executed.

The operating system has to provide several process management functions.

- **Creating new processes**. A process needs to be able to create new child processes by creating an identical copy of itself.
- **Executing programs**. A process needs to be able to replace the program it currently executes by another one. Therefore, the operating system has to provide a *loader* for executable binary programs.
- **Waiting for child processes to terminate**. A process needs to get notified when one or more of its child processes have terminated.

In order to give the impression that multiple processes are being executed simultaneously, the operating system has to divide the CPU time between the processes that are currently runnable. This process is called “scheduling”, and is performed by a separate operating system component, the scheduler.

2.1.2 Address Space Segmentation and Memory Management

The virtual address space of a POSIX process is divided into different segments. Each process contains a text segment, one or more data segments and a stack segment. The text segment contains the executable machine instructions of the program. In the data segments, global variables and the heap of the process are stored. The size of the heap is dynamic at runtime of the process. If a process requires more space on the heap, it calls the operating system to extend the size of the heap. Similarly, if heap space is no longer used, the heap size can be reduced. The stack of the process can be used to store local variables and parameters of procedures.

[2] To provide an *efficient* POSIX interface, it might be useful to incorporate POSIX abstractions into the operating system itself.

2.1.3 System Calls

System calls are black–box functions that are provided by the operating system. They allow processes to create, delete, use and modify operating system objects such as files or processes.

As system calls highly depend on the system, most programs do not call them directly. Instead, this functionality is hidden in functions that are part of the runtime library of the programming language (C library). In modern operating systems, large parts of the POSIX functionality is actually not implemented in the operating system kernel, but in the C library.

2.1.4 Interprocess Communication

In order to allow for processes to share data, various methods and abstractions have been established.

- **Shared memory** allows processes to share parts of their address space with other processes. POSIX offers two different types of shared memory: *POSIX shared memory objects* and *memory–mapped files*.
- **Message passing** permits data to be sent to other processes. POSIX provides many different ways to implement message passing. In addition to passing messages over *files*, *(named) pipes* and *network sockets*, there are also *POSIX message queue objects* and *signals* that can be sent to other processes.

2.1.5 Filesystem

The filesystem is an abstraction to hide the complexity of data storage on I/O devices. It can be thought of as a name- and storage service to provide access to data that is grouped together in “files”. The operating system provides methods to create, delete, read and write files.

The filesystem provides “directories” to organize files in a hierarchical structure, building a filesystem tree. The top directory at the root of the filesystem tree is called “root directory”. The root directory can contain files and other directories called “subdirectories”, recursively.

Files and Directories in a POSIX system

POSIX files can be divided into three categories.

- **Plain files** are ordinary binary or text files which may contain ASCII data or binary data such as executable programs.
- **Directory files** hold a list of the other files they contain.
- **Device files** are used to communicate with hardware devices. These can be divided into **character devices** such as keyboards and terminal screens, and **block devices** such as hard disks.

The filesystem provides the function "open" to convert a pathname into an operating system object called "file descriptor". The file descriptor can then be used in subsequent I/O to read data from or write data to the file or device it represents.

Access to files and devices can be restricted by "file permissions". When a process tries to open a file for read or write access, the operating system has to check if the process has sufficient permissions for this operation.

When a process is created, three file descriptors are created automatically by the operating system. These file descriptors are referred to as "stdin", which is used for input to the process, "stdout" which is used for the output of data, and "stderr" which is used for the output of error messages. When a new process is created by the *UNIX shell*, the "stdin" file descriptor is connected to the input device which controls the *shell*, normally the keyboard, the "stdout" and "stderr" file descriptors are connected to the terminal. By closing the "stdin" and "stdout" file descriptor and opening files or pipes instead, *read* and *write* calls to the file descriptors can be redirected to the files or other processes.

This so-called "I/O redirection" is a powerful tool that allows the UNIX shell to combine several simple programs to create new functionality.

2.2 Towards a LEGO–like Architecture

In traditional component-based software architecture, each software component has its own interface. Software components are created with specific interfaces in order to be combinable with other components. The combination of components often happens statically, i.e. at compile time. This component model is comparable to a *puzzle*, in that only parts with compatible interfaces can be connected and in that there is often only one way to combine the different parts into a complete system.

In object–oriented component models, this constraint is somewhat relaxed by introducing abstract interface types that can be implemented by multiple classes.

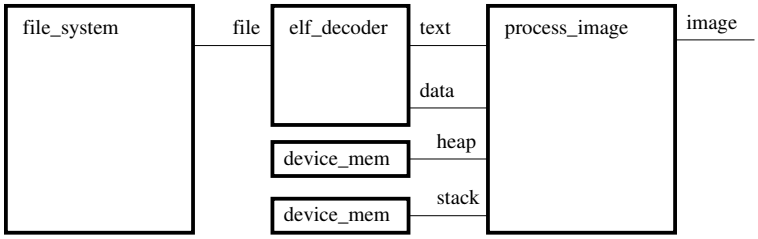

Figure 2.1: Example of interconnected bricks: Image of a user process

Nevertheless, in most object–oriented component models the number of possibilities to combine different types of components is limited by incompatible interfaces.

We overcome this limitation by using a component model similar to the LEGO toy system, i.e. by creating components that have only one, universally generic interface.

Another issue in object–oriented models is the tight integration of functionality and instantiation of components which makes it difficult to change the runtime–relationship between components.

Instance orientation[4] tries to solve this restriction by separating the functionality and runtime–relationship of components. Instead of giving components the control over their runtime relationship, this control is delegated to separate *strategy* components. The resulting anonymity of relation increases the overall modularity of the system.

2.2.1 Nests and Bricks

The generic interface of the Athomux component system is a memory abstraction called *nest* that is comparable to virtual address spaces or sparse files in UNIX. The main functionality of the nest is to provide a mapping of logical addresses into physical addresses of blocks of data.

There are a number of functions defined on the nest interface that are orthogonal to each other. The functions are implemented in software components called *bricks*. At runtime, brick instances that are connected by nests form a component system similar to “pipes and filters style” [5] as shown in figure 2.1 above.

Bricks act as transformers between their *input* nests on the left-hand side and their *output* nests on the right. In difference to most implementations of “pipes and filters style”, bricks do not have consuming semantics. They only provide a different view of the input nests at their output nests.

Looking at the example in figure 2.1, we can see a *file_system* brick providing a file at its output. The *elf_decoder* brick provides a new view to the contents of the ELF file that is connected at its input by providing access to the code and data segment inside the file at its outputs. The *process_image* brick combines these segments, along with initially empty heap and stack segments, into an address space of a user process which can be read from its *image* output.

The separation of functionality and control over runtime relation and the usage of a generic interface allows for the implementation of many small, reusable bricks that can be inserted almost anywhere in the system.

2.2.2 Elementary Operations on Nests

When a brick instance calls a nest operation at an input, the operation is delivered to and processed by the brick instance that is connected to this input. This brick instance then often calls operations on one or more of its own inputs. Since brick instances have no control over their runtime relation, operation calls on nests can be referred to as anonymous connection–oriented directional communication.

The operations on the nest interface can be divided into three groups[1].

- **invariant** operations which do not alter the state of the nest.
- **data modifying** operations which change the contents of physical blocks of data in the nest.
- **address modifying** operations which move physical blocks of data to other logical addresses in the nest.

It should be noted that executing **data modifying** operations on a nest can propagate through the system instantly and bidirectionally. Turning back to the example in figure 2.1 above, changes to the nest represented by the *file* output of the *file_system* can immediately change the contents of the nest at the *image* output of the *process_image* brick. This is possible, since the two nests can actually access the same *physical* block of data, though possibly at different *logical* addresses.

2.2.3 Generic Types

Nests contain untyped blocks of data. If data inside a nest is shared by multiple brick instances, a consensus has to be created that defines the interpretation of the data.

This can be achieved by using fixed binary data formats such as C `structs` and type conversion from blocks of data to pointers to these `structs`. However, using fixed binary types makes the system hard to extend, for changes to the data format of a type may require changes to all components that use it.

To solve this issue, the Athomux component architecture introduces a generic type system.

In this system, types can be defined at brick outputs via
`define export TYPE` typename *"string"* where *"string"* defines the contents of the type. At brick inputs, these types can then be used via
`use TYPE` typename *"string"*. If an input is connected to an output at runtime, the control mechanism has to assure the compatibility of these types.

If the definition of the exported type changes, e.g. by adding another parameter to the type, no application logic has to be changed, provided the type that is "used" is still a subset of the type that is "exported". The additional parameter is simply ignored if the brick that uses the type does not support it.

Chapter 3

Concepts of a LEGO–like Operating System

In this chapter, methods for building an instance oriented POSIX compatible operating system are discussed. As mentioned in the introduction, the focus of this work lies on a POSIX compatible filesystem and process environment to allow the execution of simple UNIX utilities.

First, we analyze the programs that the Athomux prototype should be able to execute. These programs are simple UNIX text and file processing utilities such as “cat”, “cp”, “echo”, “grep”, “ls”, “mkdir”, “mv”, “rm”, “sed”, “vi” and a simple UNIX shell called “lash”.

Needed POSIX Functionality

The system calls needed by the C library to implement the POSIX functions that are used by these utilities can roughly be divided into three categories:

- **Filesystem functions** to open, close, create and delete files, pipes and directories, to handle I/O on file descriptors, duplicate file descriptors and get status information about files and directories.
- **Process management functions** that create and destroy processes, load executable programs into address spaces, handle the size of the heap, and wait for child processes to finish.
- Functions that return **process information** about the state of the processes, such as which user it belongs to or its current working directory.

Paragraph 3.3 provides a detailed account of the implementation of these system calls.

The following section gives an overview of the realisation of traditional operating systems abstraction such as the file system and the process model in a LEGO–like operating system.

3.1 A Brick–based Filesystem

The preliminary filesystem used in this prototype was implemented by Thomas Schöbel–Theuer. The introduction to the filesystem presented here contains only the information that is relevant to understand the other parts of this work. The implementation of the bricks the filesystem consists of are described in greater detail in [2].

The Athomux filesystem is represented by the brick *fs_simple*. Files in the filesystem can be accessed as nests that are available at the outputs of the brick. The *fs_simple* brick provides the lookup and access to the files in the filesystem. The actual storage of the files is delegated to directory bricks that are instantiated inside *fs_simple*.

3.1.1 The Directory Brick

The directory brick builds several independent output nests from its input nest. The output nests constitute files, presenting a representation of a directory. To create a hierarchical filesystem, other directory bricks can be connected recursively to outputs of a directory brick. As outputs of directory bricks are essentially different views of the input nest, the input nest of the root directory brick of the filesystem contains the whole filesystem tree.

A simple example with four directories and six files is shown in figure 3.1. If an example brick instance wants to “read” from the file “/etc/passwd”, the appropriate output of *fs_simple* has to be *instantiated* and connected to an input of the example brick instance. The brick instance can then issue a read operation to this input to get the contents of the file.

It is possible to mix several different implementations of directory bricks with different characteristics inside a single filesystem. Various different implementations that use *hashes* or *b–trees* to hold and access the internal data blocks are possible.

3.1.2 Storing and Accessing File Meta Data

Athomux nests do not contain information about the data they hold. POSIX files, however, contain meta data that describe different properties of the files. These properties are e.g. the size of the file, the type of the file and the user id of its owner. In traditional UNIX–like operating systems, these properties are stored inside the inode of the file. In Athomux, which has no internal concept of inodes, the properties of a file can be stored in another section of the nest. The Athomux generic type system, as described in paragraph 2.2.3, can be used to create a representation of the data.

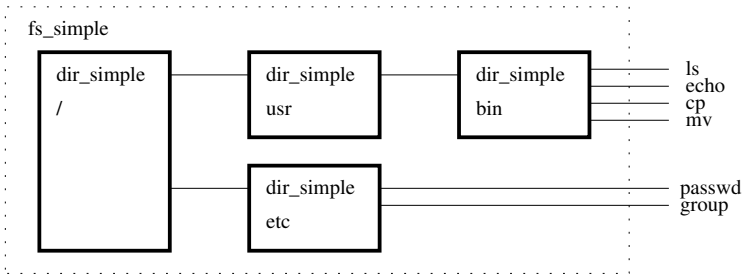

Figure 3.1: Example of the filesystem

3.1.3 The Directory Entries

A directory brick has to store the names of the files it contains. To access these name entries, the *fs_simple* brick has to provide not only access to the files that are stored in the directory bricks, but also to the meta information of the directory bricks themselves. By convention, if an output of *fs_simple* is instantiated with a trailing slash in the pathname (e.g. "/etc/"), *fs_simple* returns an output that provides access to a nest that contains the names of the directory. Otherwise, an output that represents the file, or in case of directories the whole directory as a file, is returned.

3.2 User Processes

A POSIX process consists of a segmented address space, various operating system objects that the process has access to, and information about the state of the process that is also kept in the operating system.

These operating system objects include the files which have been opened by the process. As explained in paragraph 2.1.5, the process needs to get a "file descriptor" to access the contents of files in a POSIX system. This file descriptor then serves as a "token" to reference the operating system object, the file, that it is associated with. Because the process does not own these objects, every access to it has to be made, through system calls, by the operating system.
The operating system also has to store the state of these objects. In the case of an open file this is for example the position in the file that has last been accessed, called "file pointer".

A user process in the Athomux prototype is divided into several parts.

- The **process_image** brick which provides the virtual address space of the user process.
- The **program loader** which provides the **text** and **data** segments inside the **executable file** that is currently being executed in the process.

- The **heap** and **stack segments** of the process.
- An **MMU** brick, which functions as a "device driver" for the MMU hardware that is used to map the virtual pages in the address space to actual physical pages. In the Athomux prototype, this brick is used to map the Athomux *process_images* into Linux processes.
- A **system call** mechanism that allows user processes to utilise operating system functions.

In the following paragraphs, these parts of the Athomux prototype are described.

3.2.1 The Program Loader

To be able to execute binary programs, a method is developed to create a process from an executable binary file. Therefore, a brick is required that transforms an executable binary file into a runnable process image. In the Athomux prototype, this functionality is split into two parts. First, a brick that analyzes a binary file at its input and has a dynamic number of *outputs* where nests are provided that contain the segments of the executable. Second, a brick that combines several segments into a single nest, which can be used as the address space of an Athomux process. This separation eases the support of several different executable file formats.

3.2.2 ELF Executable Files

The Executable and Linking Format (*ELF*) is a common standard for executable binary files and object code. ELF files contain an ELF header, followed by a number of sections that contain the text and data segments of the program, and information about relocation and dynamic linking [6].

Section Types

A comprehensive introduction to ELF sections can be found in [7].

There are a number of section types that need to be handled to be able to execute a statically linked binary program.

- The **.text** section which contains the program or object code.
- The **.data** section which contains the initialized global program data.
- The **.bss** (**b**elow **s**tack **s**egment) section which contains uninitialized global data.

For every section the ELF file contains a section header. This section header consists of information about the section, such as the size of the segment it contains and the memory address which it has to be mapped to. Along with the segments themselves, the program loader has to make this information available to the *process_image* brick that assembles the segment into a single address space. To accomplish this, the Athomux generic type system can be used.

In order to be able to run dynamically linked programs, additional functionality has to be implemented in the program loader to support relocation and linking at runtime.

Relocation

Shared libraries can be mapped to almost arbitrary addresses in the virtual address space of a process. Therefore, they cannot contain absolute addressing for their symbols that refer to their functions and global variables. Rather, shared libraries are compiled to “position independent code” (PIC).

To access symbols in PIC, a level of indirection is added. Since the symbols cannot be accessed via absolute addresses, they are accessed indirectly with the help of the “global offset table” (GOT). This table is created at runtime by the dynamic linker from “relocation entries” in the ELF executable. These entries contain the location at which to apply the relocation called “relocation offset” and an index into the symbol table, telling the dynamic linker which symbol is being referenced.

Although the absolute addresses are unknown at link time, the dynamic linker knows at runtime the addresses of the segments in the process and can then calculate the absolute addresses of the symbols and store them in the GOT.

Dynamic Linking[1]

Dynamically linked executables do not contain all of the code and data they need to be able to run. Some of it is held in shared libraries which can be linked into the program at runtime.

To do this, the dynamic linker has to be mapped into the address space after the executable is loaded. Instead of starting the program, the dynamic linker is then executed.

After startup, the dynamic linker finds the names of the libraries required by the program and maps their *text* and *data* segments into the address space of the process.

[1] Dynamic linking is currently not working in the Athomux prototype because of implementation issues described in 4.2.2.

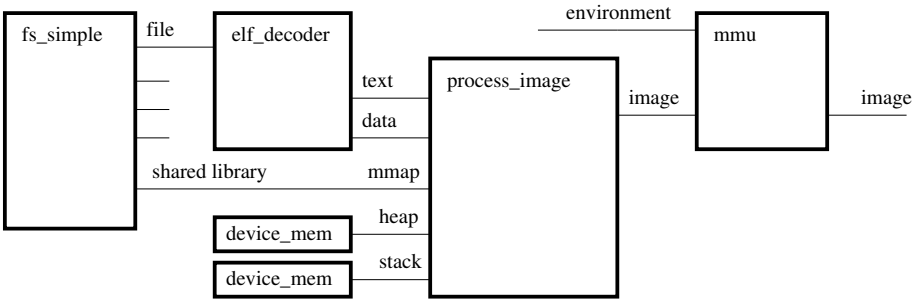

Figure 3.2: Bricks forming an executable Athomux user process

After adding the symbol tables of the libraries to the chain of symbol tables in the address space, the executable is started.

3.2.3 Address Space of a User Process

To combine the different segments into a single address space, a *process_image* brick has to be developed which combines several input nests into one output nest.

In addition to the segments from the executable file, segments have to be added that contain the heap and the stack of the process. In the Athomux prototype, these segments are provided by *device_mem_ulinux* bricks which act as a wrapper around *malloc* to allocate memory. To support dynamically linked programs and mapped files, the *process_image* has to provide additional inputs to which outputs of the filesystem brick can be connected.

An example use of a *process_image* brick is shown in figure 3.2.

When a *read* or *write* operation is called at the *image* output of the *process_image* brick instance, it has to be forwarded to the input that represents the segment accessed by the operation.

Therefore, the *process_image* brick contains a table that stores mappings of segments to virtual addresses in the address space. This information can be passed on by the *elf_decoder* brick using the Athomux generic type system.

3.2.4 Executing a Program

In possible native Athomux implementations, the execution of an Athomux address space that contains a user process takes place in an *MMU* and a *CPU* brick [1]. The *MMU* brick provides the mapping of the data in the *process_image* to

physical pages in system memory. It hence acts as a device driver for the page table of the hardware MMU. The *CPU* brick acts as a driver for the CPU. A scheduler component has to switch the CPU between the *MMU* instances to provide multitasking.

In the Athomux prototype, Athomux user processes and the Athomux operating system run as ordinary Linux processes that do not have access to the hardware CPU and MMU directly.

To solve this, we implement a brick *mmu_ulinux* that provides the mapping of an Athomux *process_image* into a Linux user process. The scheduling of the processes is then done by the Linux operating system, so *CPU* and *scheduler* bricks are not required.

The implementation of this approach is explained in paragraph 4.1.

3.3 POSIX Functionality

At the beginning of this chapter, three groups of system calls were identified that are needed to implement various *UNIX* utility programs which provide basic functionality such as a system shell.

In this paragraph, first, a mechanism is presented to provide a POSIX system call interface in a LEGO–like operating system. Then, the various system calls and the concepts required for their implementation are presented.

3.3.1 A POSIX System Call Interface

The system call interface is placed inside the *MMU* brick which itself contains various bricks. System calls are received inside the *syscall* brick that contains the processor dependent binary system call interface to the user programs.

After the system call is received by the *syscall* brick, it is packaged inside a generic “syscall type” and sent to be processed by other bricks, depending on the type of the system call.

The responsibility of system call processing is divided into three bricks.

- A *strategy* brick that processes system calls that need new bricks to be instantiated or new brick connections to be made.
- A brick that handles system calls that provide I/O on file descriptors.
- A brick that handles system calls that provide information about the process.

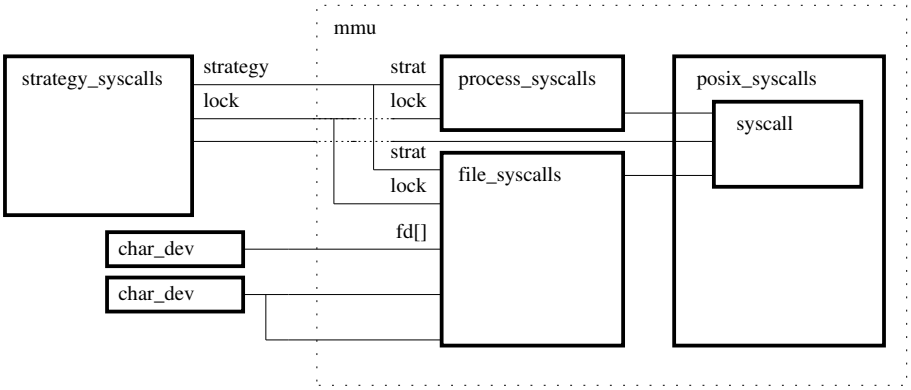

Figure 3.3: The system call interface

It should be noted that the grouping of system calls that was made at the beginning of this chapter and the responsibility of system call handling are not equivalent.

The first grouping is a division by POSIX semantics of the system call, the second grouping is a division by implementation logic in a brick-based operating system. The bricks that are responsible for system call handling are sketched in figure 3.3.

To provide an insight into the concepts behind the mapping of POSIX functionality to instance oriented principles in a brick-based operating system, some of the most interesting system calls are described in the next paragraphs.

3.3.2 Process Management Functions

This group of system calls entails functions which

- **Create** and **delete** processes.
- **Load** and **execute** programs.
- **Increase** or **decrease** the size of the heap.

With the exception of the modification of the size of the heap, all these functions need to create or destroy brick instances and establish new connections to these instances. These functions have to be implemented at the strategy level.

Creating a new Process by forking a Process

The only way to create a new POSIX process is to duplicate an existing process with the system call “fork”[2]. The created child process differs from its parent

[2]This also means that the creation of the first process is outside the scope of POSIX.

process only by its process id.

One possibility of implementing this is to create a new *MMU* brick instance and a new *process_image* brick instance with identical contents as the *process_image* brick instance of the parent process. Although, most of the time, a process is only forked to execute another program. This means that immediately after the process is duplicated, the address space is cleared and another program is loaded.

Since duplicating an entire address space is very expensive, it is better to continue to use the old address space for *both* processes and to copy only those pages that are changed by one of the processes after the fork.

This strategy is called "copy on write" (COW) and is implemented in most UNIX–like operating system for better performance and lower complexity of the *fork* system call.

We implement this behaviour by creating a new *MMU* brick instance for the child process, and connecting a *COW* brick instance to the *process_image* and the *MMU* bricks of the two processes. When a process calls a *write* operation on an output of the *COW* brick, the referenced data block is duplicated inside the *COW* brick instance that has to keep track of modified blocks. For fast lookup of modified blocks, the copied blocks of data are stored inside a hash table. If a block that is referenced by a *read* operation is currently unmodified by either process, the operation is simply forwarded to the *process_image* brick instance. When the block is already modified, the operation is not forwarded but processed locally inside the *COW* brick instance.

By forking a process, the open files of the parent process are inherited by the child process and the file descriptors are subsequently shared between them.

This is implemented by connecting all existing *file descriptor* brick instances the parent process is connected to, to the newly created *file_syscalls* brick instance of the child process.

The resulting brick network containing a shared file descriptor is shown in figure 3.4.

Deleting Processes

When a process is about to quit, it calls the *exit* system call. At exit of a process, all open files have to be closed. The process can not be removed completely as the "return value" of the process must be stored until the parent of the process requests it.

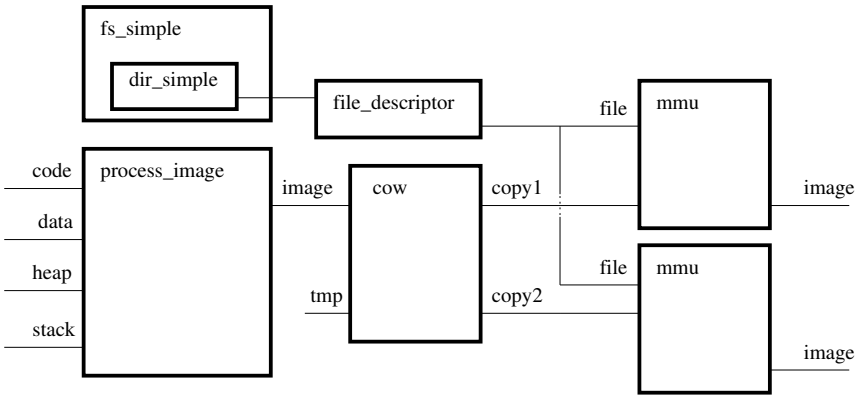

Figure 3.4: Two processes sharing an open file after *fork*

Executing new Programs

When a process calls *exec* to execute a new program, the address space of the process has to be cleared and the new program has to be loaded and executed.

First, it has to be checked whether the file that is to be executed actually exists. If the process was created by *fork*, a new *process_image* has to be created and connected to the already existing *MMU* brick instance to which the *COW* brick instance has been connected before.

Then, a new *elf_decoder* brick is instantiated and connected to the executable file from the filesystem brick *fs_simple*. The inputs of the *process_image* are then connected to the *text* and *data* outputs of the *elf_decoder* brick instance as described in paragraph 3.2.

The strategy component that handles the system call has to close all the files in the process that have the “close on exec” flag set before the new program can be started.

Handling the Heap

When a process is first started, its heap is empty. To allocate space on the heap in order to store data structures, POSIX provides function “malloc”. This function is typically implemented in the C library. Various methods of implementing “malloc” exist. For instance, it is possible to map an anonymous file with the needed size to an unused address in the address space for every call to “malloc”.

This method is easy to implement but highly inefficient because it requires to perform one system call per “malloc”.

Another, more efficient method that is used by C libraries is to allocate space on the heap and portion parts of it out when “malloc” is called. Although the implementation is more difficult, it is generally preferred because it does not need to issue a system call for a “malloc” when there is still enough free heap space left.

The operating system only needs to provide a system call to allow the heap to grow and shrink dynamically. The implementation of this system call “brk” is generally very easy.

In the Athomux operating system, space in the *process_image* of the user process has to be allocated or freed in the heap segment, depending on whether the heap size should be increased or decreased.

3.3.3 Operations on Files

This group of system calls can be divided into system calls that perform I/O on file descriptors, and system calls that handle the opening and closing of new file descriptors.

The first group can be implemented by the *file_syscalls* brick, the second group requires new bricks to be instantiated and connected and therefore has to be implemented at the strategy level.

Opening Files

If the file that the process wants to open exists in the filesystem, it is instantiated as a new output in the filesystem brick.

A free entry in the file descriptor table, represented by inputs of the *file_syscalls* brick instance, has to be found. A new *file_descriptor* brick has to be instantiated and connected to the output of the file system that represents the file, and the free input of the *file_syscalls* brick instance.

The number of the newly allocated file descriptor can then be returned to the process. In case of failure, e.g. if the file does not exist or if there is no more space in the file descriptor table, the appropriate error code is returned.

Closing File Descriptors

A file descriptor can be closed by disconnecting and deinstantiating the *file_descriptor* brick instance that it is associated with.

As file descriptors can be shared between parent and child processes, removing and disconnecting the *file_descriptor* brick instance closes the file descriptor for

all processes that have access to it, which is actually the POSIX semantic of the "close" function.

Duplicating File Descriptors

Duplicating a file descriptor means creating a new file descriptor that shares the file pointer, file locks and the access mode of the referenced file with the old file descriptor.

In the Athomux prototype, this is done by using the already existing *file_descriptor* brick instance as the new file descriptor and connecting it to an additional input of the *file_syscalls* brick.

If a duplicated file descriptor is closed, the associated *file_descriptor* brick may only be removed if all connections to the *file_syscall* brick have been disconnected before.

I/O on File Descriptors

I/O on file descriptors is implemented by the "read" and "write" system calls. In the Athomux prototype, these functions are processed by the *file_syscalls* brick where all file descriptors of the process are connected.

The system calls can be translated to equivalent Athomux nest operations that are then called on the appropriate input. When reading from a nest that represents a file, it is required to examine if the read request reaches the end of the file.

Similarly, when writing to a file, the size of the file has to be adjusted if the write request crosses the end of the file.

3.3.4 Interprocess Communication and Locking

This paragraph explains methods how interprocess communication can be implemented in a LEGO–like system.

However, interprocess communication other than I/O redirection via pipes is not required by the Athomux prototype in order to execute the standard UNIX utilities. Hence, these concepts are currently not implemented in the Athomux prototype and untested.

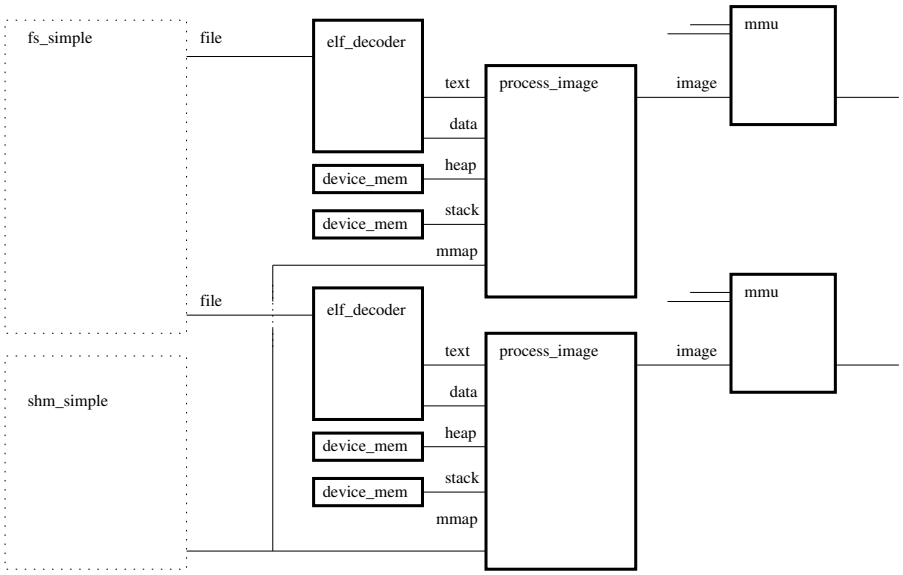

Figure 3.5: Interprocess communication with the usage of shared memory

Shared Memory

To allow processes to share data, POSIX provides shared memory objects. These objects can be created and mapped into the address spaces of unrelated processes.

POSIX shared memory object are similar to files, with the exception that they do not appear inside the filesystem hierarchy.

To support POSIX shared memory, we propose to implement a naming service such as the already existing filesystem brick. This *shm_simple* brick contains the shared memory objects and provides access to them at its outputs.

The proposed architecture is sketched in figure 3.5.

Waiting for the Termination of Processes

The system call “wait” can be used by processes to wait for termination of one of their child processes.

The implementation in the Athomux prototype uses locks that are provided by the Athomux component system. When a process is started, it locks a part of the memory that is unique to the process.

To wait of the termination of the child process, the father process tries to get the lock of the memory area that is held by the child. When the child terminates, the lock is freed and the parent process can now get access to it.

Waiting for more than one child process to finish is currently not implemented in the athomux prototype. With this approach, disjunctive waiting for locks would be required which is currently not possible with the locks that are provided by the Athomux component system.

Chapter 4

Implementation of Key Concepts

Having established the concepts that are required to implement a LEGO–like operating system, we need to provide two additional mechanisms to be able to run the user processes of the Athomux prototype in a Linux environment.

- The implementation of the *mmu_ulinux* brick that is used to map an Athomux address space provided by the *process_image* brick into the address space of a Linux process.
- The implementation of the system call interface to call Athomux system calls from Linux processes.

These mechanisms are introduced in the following paragraphs.

4.1 Processes in the Athomux Operating System Prototype

The Athomux operating system prototype and its user processes are running as ordinary Linux user processes in a Linux environment. To allow Athomux user processes to be executed, a method has to be created to be able to execute an Athomux nest – which contains the address space of the Athomux user process – in a Linux process.

4.1.1 Mapping Athomux Address Spaces into Linux Processes

A Linux process consists of a virtual address space containing text, data and stack segments. The text segment contains the Linux binary program that is currently being executed in the process. The binary program is loaded by mapping the file containing the program into the address space of the process. To do this, the Linux kernel provides the system call *mmap*, which is able to map Linux files into the address space at arbitrary, currently unused locations. To allow the mapping

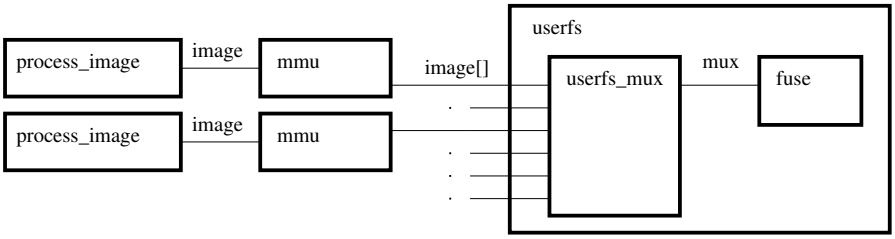

Figure 4.1: Using FUSE to access Athomux nests as Linux files, Athomux view

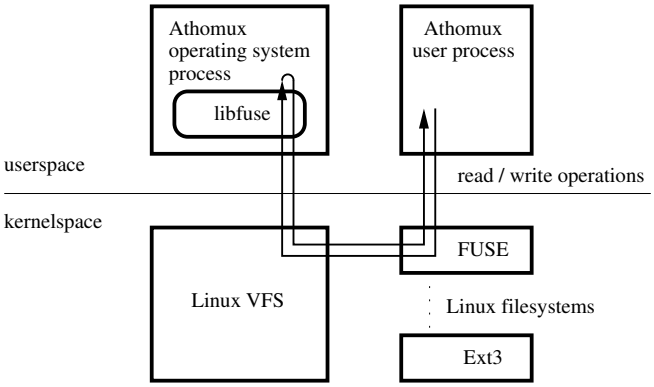

Figure 4.2: Using FUSE to access Athomux nests as Linux files, Linux view

of an Athomux nest into the address space of a Linux process, instead of a Linux executable, a method is needed to export the nest as a file into the Linux filesystem. The file representing the Athomux nest can then be mapped via *mmap* at the proper location into the address space of the process.

To export an Athomux nest into a file, we use the “Filesystem in Userspace” (FUSE) Linux kernel module [8]) . FUSE provides callbacks that hook into the Linux Virtual Filesystem Switch (*VFS*), allowing Linux userspace programs to implement a virtual Linux filesystem that can be mounted into the Linux filesystem tree.

The Athomux operating system process has to implement the filesystem calls *read* and *write* in the virtual FUSE filesystem, by calling the appropriate nest function on the *image input* of the respective Athomux *process_image* brick. This functionality is implemented in the *userfs* brick, to which the *process_image* bricks of the Athomux user processes are connected to, as shown in figure 4.1.

The individual address spaces of the Athomux user processes appear as files, with their process id as the name, in the Linux filesystem. Calls to *read* and *write* on open files from the FUSE filesystem are then redirected, through the Linux *VFS*, to the Athomux process as shown in figure 4.2.

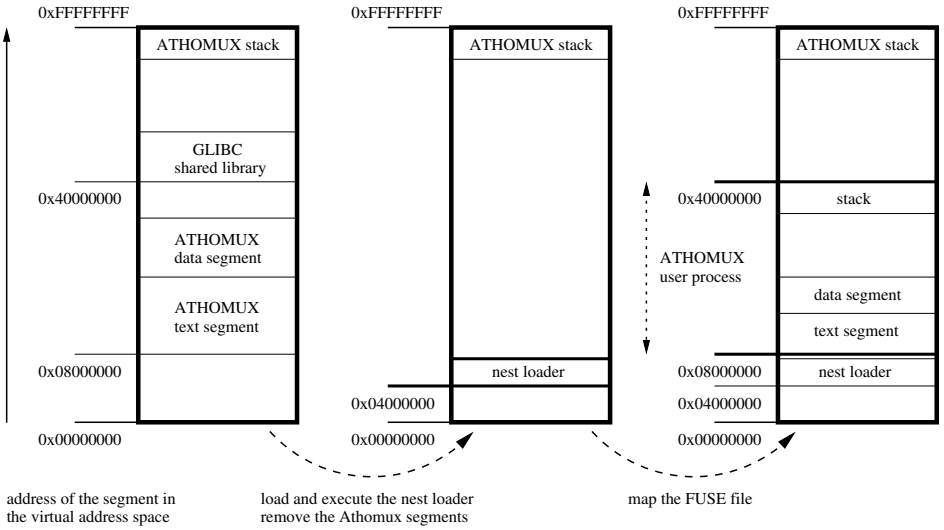

Figure 4.3: Mapping an Athomux nest into a Linux process

4.1.2 *Fork* and *Exec* in the Athomux Prototype

New POSIX processes are created by construction of an identical copy of another process by the *fork* system call. The program executed by the process can then be replaced by another program by calling the *exec* system call.

To create a new Athomux user process, a new Linux process has to be created accordingly. When an Athomux user process calls *fork*, a new Athomux user process – consisting of a *process_image*, an *mmu* and various other bricks as shown in 3.3.2 – has to be instantiated. A new Linux process is then created by *fork*ing the Athomux operating system process. The Athomux operating system program has then to be removed from the newly created Linux process and replaced by the file from the virtual FUSE filesystem containing the address space of the newly created Athomux user process.

Since a Linux program that is currently running cannot remove itself from the address space, a separate *nest loader* program is needed. The nest loader program is loaded to a free space at the bottom of the newly created Linux process and executed. First, the loader has to remove the Athomux operating system program from the address space. It them opens the file from the virtual FUSE filesystem, and maps it to the location in the address space that was formerly occupied by the Athomux operating system program. The nest loader then sets the appropriate base pointer and stack pointer and transfers control to the Athomux user program, which continues execution after the return of the *fork* system call. This process is outlined in figure 4.3.

The implementation of the *exec* system call, which replaces the currently running

program with another one, is similar to that of the *fork* system call. Instead of duplicating the *process_image* brick, as was needed by the *fork* system call, a new *process_image* brick is created and connected to an *elf_decoder* brick as is shown in figure 3.2. The *process_image* is then exported to the Linux filesystem by the *userfs* brick. A new Linux process is created by *fork*ing the Athomux operating system process. The nest loader is then loaded into the address space and executed. In contrast to the implementation of the *fork* system call, the nest loader has to copy the environment variables from the parent process, set the command line arguments with which the process shall be created, and set the instruction pointer to the entry point of the executable.

4.1.3 Limitations of the Nest Mapping

The nest mapping that has been introduced contains several limitations.

As described in the previous paragraphs, the address space of an Athomux user process is mapped into the Linux process with the help of FUSE and *mmap*. To be able to use the mapped file for bidirectional communication between the user process and the Athomux operating system process as is necessary for system calls, the following requirements would have to be met.

- (1) Changes that are made to memory locations in the address space where a file is mapped have to be written back to the file instantaneously.
- (2) Changes that are made to files that are mapped into the address space with *mmap* have to appear immediately inside the address space.

Unfortunately, both requirements are not fulfilled by *mmap*.
Requirement (1) is enforceable by explicitly calling *msync* after every change to the address space (or at least before issuing a system call).
Requirement (2) cannot be enforced easily. A workaround to be able to communicate changes on the file back to the process could be to stop the process by a jump to the nest loader, unmapping the virtual file, mapping it back in immediately and then to continue execution at the former location.

4.2 The System Call Interface

In typical UNIX-like operating systems, the implementation of POSIX functionality is divided between the operating system kernel and the C library. User programs usually don't call kernel functions directly, they call functions in the C library which then call kernel functions if necessary. There are many advantages in putting as much functionality into the C library (and therefore into userspace) as possible:

- The operating system kernel is hard to debug, so it has to be kept small.
- Many parts of POSIX can be developed operating system independent.
- Calls from userspace into kernelspace are expensive and thus have to be minimised.

Since POSIX functions are relatively high-level, often share functionality or don't have to use operating system functions at all, the number of system calls that a typical UNIX-like operating systems has to support is substantially lower than the number of POSIX functions.

To call a Linux kernel function, the C library loads the CPU registers with the appropriate values and issues an interrupt. The kernel then processes the interrupt and returns the results to the user process.

In the Athomux prototype, another mechanism has to be used, since processing of the system calls has to be done by the Athomux operating system process and not by the Linux kernel. There are different ways to achieve this, most of which include the requirement to make modifications to the Linux kernel. E.g. it would be possible to install another system call handler in the kernel that passes control to the Athomux operating system process. For reasons of simplicity, a solution that works completely in userspace is clearly preferable. We chose to change the system call mechanism of an existing C library and link Athomux user programs against this library. As the C library, uClibc [9] was used, a standard library for embedded systems that use Linux as operating system.

4.2.1 Adaption of an Existing C Library

Instead of issuing interrupts to process the system calls, the modified C library writes all parameters of the system call to a Linux pipe. The Athomux operating system process reads the system call information from the other side of the pipe, processes the call and writes the results back. This system call mechanism as is drafted in figure 4.4 is similar to the message-based system call mechanism found in DragonFly BSD [10].

Since the Linux operating system kernel has complete read and write access to the address space of the processes, most system calls to the Linux kernel are "call by reference". The function calls of the C library have to be converted to "call by value", since the Athomux operating system process does not have access to the actual content of the address space of the user processes because of the limitations of the nest mapping mechanism shown in 4.1.3. This obviously increases the complexity of a system call, but performance is of minor concern to the implementation of this prototype.

Aside from the changing of the parameter passing method, almost all system calls can be adopted unchanged. At a few locations however, additional parameters

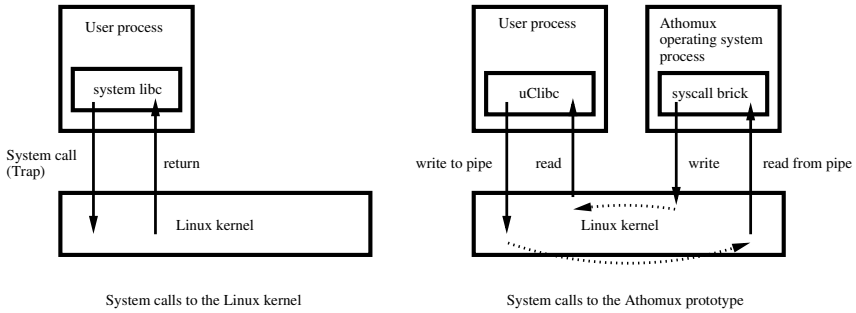

Figure 4.4: System calls in Linux and the Athomux prototype

have to be introduced. This happens when the Linux kernel can get additional information about the processes, which the Athomux operating system process cannot obtain. This is the case e.g. with the *fork* system call that creates a new child process that continues its execution at the location after the system call occurred in the father's process. The Athomux operating system process needs additional information about the current status of the user process that issued the system call, namely the base pointer, the stack pointer and the instruction pointer to continue execution of the child process at the right position.

4.2.2 Dynamic Linking Issues

To support dynamic linking as explained in paragraph 3.2.2, the runtime linker has to be able to map shared libraries into the address space. This has to be done by a system call like *mmap* that allows the mapping of files into the address space at arbitrary addresses.

To execute a system call with the mechanism proposed above, the user process already needs functionality from the C library to communicate with the Athomux operating system process via the system call pipe. This means that at least the C library has to be linked statically into the Athomux user programs.

4.3 State of Development

With the mechanisms outlined, it was possible to create an operating system prototype built on a brick component system. This prototype is able to execute some simple UNIX utilities for file management such as "ls", "cp" and "mv", and some more complex programs such as a system shell and a text editor. These utilities were taken, with very few changes, from the BusyBox project [11] that aims to develop standard UNIX utilities for embedded systems.

There are still a few drawbacks in the prototype to consider:

- Most of the system calls that are supported are not implemented completely, the *open* system call alone can take up to 11 different flags and return 13 different result values. Full POSIX compatibility is nowhere near possible in the scope of this work because of the complexity of the standard.
- A method to support signal handling has not yet been developed. The issues are not so much on the Athomux side, but in communicating asynchronous signals to the Linux processes without using the Linux signal mechanism itself.
- The implementation of the *fork* system call is currently only working correctly if *exec* is called immediately thereafter[1]. This is due to difficulties with the correct restoring of the state of the newly created process after fork.
- The support of shared memory and threads would require additional work because of the limitations of the nest mapping described in paragraph 4.1.3.

It should be noted that almost all of the problems that occurred during this work are due to the "simulation" of the Athomux operating system and its user processes in Linux processes. A native implementation of Athomux that uses the concepts that were developed during this work should not have these issues.

[1] Which is actually the functionality of the system call *vfork*.

Chapter 5

Conclusion

5.1 Summary

We have shown that traditional UNIX–like operating systems can be built by applying methods of instance orientation. We have designed concepts to build core operating system abstractions such as a process model and a filesystem using a LEGO–like brick component system. A mechanism was introduced that provides a way of implementing a POSIX system call interface in an operating system based on bricks. In the progress of this work, we have developed means that allow the implementation and testing of new ideas in operating system design in a Linux userspace environment.

Using these concepts we have successfully implemented an operating system prototype called Athomux that is able to run simple utility programs such as a UNIX shell.

5.2 Unresolved Issues

We have concentrated on the development of concepts that are needed to be able to run standard UNIX applications, the integration of the filesystem and the development of a process model.

Because of this, concepts such as interprocess communication and threads were not covered in depth. Further research in this area is needed.

The implementation of the Athomux prototype contains several interesting problems that could only be addressed partially. Because of these limitations, it was not possible to implement operating system abstractions such as threads or shared memory. It should be noted that these limitations are not caused by the Athomux architecture, but rather by the implementation of the *mmu_ulinux* brick that provides the mapping of Athomux nests into Linux address spaces.

5.3 Future Work

First tests with an implementation of a *remote* brick[1] that provides the sharing of nests across a network, have supported the assumption that distributed and network transparent operating systems can be developed easier and faster than with existing approaches.

Future research in this area is particularly interesting since network transparency can be implemented at the strategy level by inserting *remote* bricks at almost arbitrary locations in the system.

A possible improvement would be the development of an *MMU* brick that runs in Linux kernelspace. Such an implementation would not require the userspace filesystem to map Athomux nests into Linux user processes and the system call mechanism presented in this work.

This would solve the limitations of the current implementation of the prototype that are described in paragraph 4.1.3 and 4.2.2 and provide the possibility to implement POSIX threads, shared memory and dynamic linking.

5.3.1 Missing Functionality for POSIX Compliance

To test POSIX compliance, the C library used in this work (uClibc) provides a large test suite that contains regression tests.

Although these tests are primarily developed to locate differences between uClibc and the GNU C library (glibc) implementation, they can also be used to test the compliance of the Athomux prototype to POSIX standards against other implementations of UNIX–like operating systems such as Linux. Another test suite that provides conformance, functional, and stress tests can be found in [12].

To provide meaningful test results, too few functions of the POSIX standard could be fully implemented during the time of this work.

5.3.2 Athomux as a Standalone Operating System

To be able to run Athomux as a completely standalone operating system, much more work is needed.

The better part of the required effort is the development of device drivers. Instead of having to develop device drivers from scratch, methods could be developed to automatically extract device drivers from the Linux kernel to make them available as Athomux bricks.

[1] The *remote* brick was implemented by Hardy Kahl for his not yet published diploma thesis.

5.3.3 Researching an Alternative Interface Design

The core of the POSIX interface has evolved from the monolithic kernel design of early UNIX operating systems.

We believe that POSIX is not the ideal interface for application development in operating systems that are built on instance oriented principles. Instead, it should be analyzed how application programs can benefit from using the nest and brick abstractions to utilize operating system functionality.

Appendix A

Software Components

To install and use the Athomux prototype that is presented in this work, some dependencies have to be installed first.

The Filesystem in Userspace

The “Filesystem in Userspace” (FUSE) is a Linux kernel module that provides hooks into the Linux VFS to allow the implementation of filesystems inside Linux userspace processes.

In this work, FUSE version 1.3 was used to implement the mapping of Athomux nests into Linux processes as described in chapter 4.1. The FUSE kernel module can be downloaded from the website of the FUSE project [8].

The C Library

As C library that provides most parts of the POSIX functionality, uClibc version 0.9.26 is used [9]. We decided to use uClibc instead of the GNU C library (glibc) because it is relatively small and can be configured to a great extend to remove unneeded functionality.

A new architecture was added to uClibc called “athomux_linux” In this architecture, the system call mechanism was changed to use UNIX pipes instead of the “trap” function as described in paragraph 4.2.

Unix Utilities

Standard UNIX utilities which can be used in the prototype are provided by the BusyBox project [11]. In this work, BusyBox version 1.00-pre7 was used.

The Makefile had to be changed to compile and statically link the programs against the Athomux C library. Some minor changes to the shell had to be made to remove functionality that is not yet provided by the Athomux prototype.

Bibliography

[1] SCHÖBEL-THEUER, THOMAS: *Eine neue Architektur für Betriebssysteme*.
Unveröffentlichtes Manuskript einer Monographie, erhältlich auf Anfrage bei `ts@athomux.net`, 2003.

[2] SCHÖBEL-THEUER, THOMAS: *Skizze einer auf nur zwei Abstraktionen beruhenden Betriebssystem-Architektur: Nester und Bausteine*.
Arbeitspapier und Vortrag auf dem Herbsttreffen der GI, Fachgruppe Betriebssysteme, Berlin, 7. – 8.11.2002.

[3] RITCHIE, DENNIS M.: *The Evolution of the Unix Time-sharing System*.
AT&T Bell Laboratories Technical Journal, 63(6):1577–1593, 1984.

[4] SCHÖBEL-THEUER, THOMAS: *Instance Orientation: a Programming Methodology*.
In *to appear in SEA 2004*. IASTED conference proceedings, Cambridge, MA, November 2004.

[5] SHAW, MARY and DAVID GARLAN: *Software Architecture, Perspectives on an Emerging Discipline*.
Prentice Hall, 1996.

[6] LEVINE, JOHN R.: *Linkers and Loaders*.
Morgan Kaufmann Publishers Inc., 1999.

[7] YOUNGDALE, ERIC: *Kernel Korner: The ELF Object File Format by Dissection*.
Linux Journal, 1995(13es):15, 1995.

[8] SZEREDI, MIKLOS: *Filesystem in Userspace (FUSE)*, 2004.
http://fuse.sourceforge.net/.

[9] *The uClibc project*, 2004.
A C library for embedded systems. http://www.uclibc.org/.

[10] HSU, JEFFREY M.: *The DragonFlyBSD Operating System*.
Technical Report, AsiaBSDCon, 2004.
http://www.dragonflybsd.org/goals/userapi.cgi.

[11] *The BusyBox project*, 2004.
UNIX programs for embedded systems. http://busybox.net/.

[12] *Open POSIX Test Suite*, 2004.
http://posixtest.sourceforge.net/.

Declaration

All the work contained within this thesis,
except where otherwise acknowledged,
was solely the effort of the author.
At no stage was any collaboration
entered into with any other party.

_____

(Florian Niebling)