Evaluating POSIX-Compatibility on Top of a Component-Based Operating System

Studienarbeit

Stefan Götz
System Architecture Group
Universität Karlsruhe
sgoetz@ira.uka.de

2001.12.07

Abstract

This study thesis investigates how the GNU C library can be ported to the SawMill operating system identifying and proposing solutions for relevant design issues, and describing details of their implementation.

Contents

1 Introduction

SawMill is a multi-server operating system based on the L4 micro kernel. Currently, there is a small number of applications available. It is desirable to support a wide variety of applications, both to establish SawMill as a productive system and to evaluate the system. When using existing applications for this purpose we can benefit from a significantly reduced implementation effort, and the possibility to directly compare application performance of SawMill and other systems.

Running existing applications on top of SawMill requires an emulation layer which maps SawMill's component based services to the application's native operating system interface. We aim to create a POSIX layer on top of SawMill by modifying the GNU C library, which itself is designed as a POSIX interface on top of underlying operating systems.

This thesis investigates how the GNU C library can be ported to SawMill. This includes identifying the relevant design issues and proposing solutions for them as well as a description of how the changes to the GNU C library were implemented.

1.1 Motiviation

One goal of SawMill is to run existing standard applications on top of it. Such a setting would allow us to examine the employed concepts from an architectural point of view as well as their implications on performance. This can tell us, how applications need to be designed or modified so they can benefit most from SawMill's component structure.

1.2 Goal

Our basic goal is to create a – initially partial – POSIX emulation environment on top of SawMill.

1.2.1 The POSIX Emulation

We chose POSIX as the platform to emulate for the following reasons:

- POSIX is both a well established and widely used standard. A large application base is available.
- Many applications are open source allowing us to inspect and modify them where necessary.
- The POSIX interface is usually provided via a C library instead of beeing fully implemented in the operating systems. This gives us the opportunity to use the C library itself as an emulation layer instead of creating one from scratch.

The GNU C library was chosen for similar reasons: it is very commonly used by the programs we are interested in, published as open source and its structure is well-suited for porting to new operating systems.

It is clearly beyond the scope of this thesis to provide propositions or implementations for all GLIBC features on top of SawMill. Therefore, we narrowed our focus on a subset of its functionality which supplies a reasonable feature base for applications and reduces the implementation effort to a manageable amount.

In order to reach a very basic level of functionality the following GLIBC mechanisms need to be implemented: support for the POSIX memory

model and basic file and terminal I/O. This feature set gets a simple 'Hello World' program up and running.

Dynamic linking and loading is interesting for us as we want to learn about the effects of code sharing on native SawMill programs. Since this functionality can be achieved with an acceptable amount of extra work based on the features mentioned before, we decided to support it both for standard programs linked with the modified GLIBC and for SawMill components.

To provide the basic interface the emulation also needs to include process management, process creation via fork, signals and terminal related functions. We decided to flesh out the emulation to the extent that a simple shell can be linked and run with the modified library and users can start other applications linked with it.

1.2.2 Specific Objectives

We aim at emulating the following features:

- Memory Management: `malloc()` via `brk()`
- File and Terminal Access: file name resolution, `open()`, `close()`, `read()`, `write()`, `mmap()`, `select()`
- Signals: `raise()`, `kill()` and all other signal related POSIX functions
- Application Execution: `fork()`, `exec()`
- Dynamic Linking and Loading

1.2.3 Summary

Based on the GLIBC, our primary goal is an emulation framework for fundamental POSIX facilities, which relies on SawMill's system services. The interfaces to the SawMill services need to be designed in an extensible fashion so the implementation can initially cover the most relevant aspects and later be enhanced step by step.

We expect that this process leads to a re-evaluation of existing SawMill component interfaces and yields valuable insights for future component design.

2 Related Work

The importance of the POSIX standard is underlined by the number of solutions developed to achieve POSIX compatibility on top of non-POSIX systems. This chapter gives a brief overview of the different approaches that have been implemented to achieve this goal.

2.1 POSIX Emulation on Micro Kernels

Component based and multi server operating systems like SawMill cannot rely on a well-established interface tailored to their demands and strengths. To run standard applications they need to emulate more traditional OS interfaces like POSIX or they are explicitely designed to provide a UNIX like interface to applications.

2.1.1 The GNU Hurd

The GNU Hurd is a multi-server operating system based on the Mach kernel. The GNU C library serves as an intermediate layer between the Hurd system components and applications providing a POSIX compatible operating system interface. It is to be regarded as part of the operating system with fundamental POSIX semantics (e.g. users) being integral to Hurd components.

The Hurd allows users to exchange certain system components without compromising the security and stability of the complete system. It extensively follows the UNIX concept of mapping all system objects into the global file name space.

The performance of the Hurd is substantially impacted by Mach's slow user-to-user IPC and expensive user-level page fault handling [CB93][Lie95].

2.2 POSIX Emulation on Windows Systems

The attempts of emulating the POSIX interface on Windows are mainly targeted at reducing the effort necessary for porting existing programs from UNIX to Windows systems and have thus yielded a variety of products based on two fundamentally different emulation concepts.

2.2.1 Emulation Libraries

RedHat's Cygwin [Noe98] and UWin [Kor97] developed by AT&T as well as NuTCracker from Datafocus and Portage from Consensys are similar in their basic design. They mainly consist of a dynamic link library which emulates the POSIX functions. Applications that are linked against it execute in the standard Win32 environment thus also allowing the invocation of Win32 functions from such applications.

The basic emulation strategy is similar to our concept of implementing system services in a library which makes use of system components. These products therefore face similar problems as our approach and it should prove useful to investigate their design choices for specific emulation issues.

Files are emulated based on the underlying Windows file system. Apart from problems like the different name space layouts and different line termination characters the emulation is straight forward.

Signals are emulated via a signal thread running in the address space of the application. The library's emulated system calls are protected by synchronization mechanisms to detect or prevent the occurrence of signals in these functions.

The `fork()` Function is implemented by creating a new address space and the management data structures for the child in the emulation library, copying the contents of the parent's to the child's address space and restoring the parent's execution context in the child. Since this approach is inefficient due to the lack of copy-on-write semantics and multiple task switches between the parent and the child, and the emulation of the `exec()` function is faced with similar problems, both Cygwin and UWin implement a spawn family of functions which resemble task creation on Win32 system much closer and perform considerably better than the `fork()`/`exec()` function pair.

The `select()` Function in Cygwin relies on a Win32 equivalent for sockets. When applied to file descriptors, a thread is created for each descriptor polling for activity. This causes considerable resource consumption and a more efficient method should be implemented in SawMill.

2.2.2 Emulation Subsystems

On WindowsNT, a kernel provides essential hardware independent services to functional subsystems (like security and I/O) and environment subsystems (e.g. Win32, Microsoft POSIX, OS/2) which implement higher level functionality accessible to applications. Since the POSIX subsystem implemented by Microsoft is very restricted, OpenNT [Wal97] was developed by Softway Systems to create a more complete UNIX environment on top of the WindowsNT kernel.

OpenNT mainly consists of the UNIX subsystem itself, a terminal session manager for each terminal group and a dynamic link library interfacing to the UNIX subsystem and handling some system service requests directly.

Unfortunately, there is no detailed information on the actual implementation. It can be assumed though that most services are provided by the UNIX subsystem and OpenNT is effectively similar to a monolithic kernel in contrast to the libary approach.

3 Basics

This chapter covers the fundamental paradigms of the systems involved in our emulation goal: POSIX, the GLIBC, L4 and SawMill, .

3.1 POSIX

The POSIX standard defines a portable operating system interface. Compared to L4 and SawMill, it mainly covers higher level concepts which are described in this chapter.

3.1.1 Files

POSIX characterizes files as objects that can be written to or read from or both. Files have certain attributes, including access permissions and type and are usually named. File types include physical files, IPC facilities and devices.

Depending on the file type, files can only be accessed via a handle returned by the `open()` function which performs access control. After opening a file, applications use the `read()` and `write()` functions to access the file's contents. The operating system copies the data to be read from the file to a memory buffer specified by the application (for write, the data is copied from the buffer). Depending on the file type, the `mmap()` function allows mapping a file into virtual memory so the application can access it by directly reading from or writing to memory.

3.1.2 Processes

POSIX calls an execution context a *process*. A process executes in an address space. Originally, address spaces held only one process. The *pthreads* (POSIX-threads) specification introduces the thread as execution context. A single address space can be occupied by multiple threads. Linux on the other hand allows processes to share page tables, i.e. address spaces. These extensions are not covered here, we deal with one process per address space.

New processes can be created by any existing process via the `fork()` function. An execution environment for the new process is created by duplicating the address space contents of the creator so that the states of parent and child process are largely identical when the `fork()` returns. The new process starts to execute as if itself just returned from `fork()`.

The `exec()` function allows processes to discard the contents of their address spaces to load and run a specified executable.

3.1.3 Address Spaces and Memory Management

POSIX specifies a flat address space model. Address spaces are created together with new processes by calling the `fork()` function. The new address space is initially constructed as a duplicate of the invoker's address space.

POSIX address spaces are described by file mappings. A file mapping is an association of a part of an address space with a portion of a file. Accesses to virtual memory are effectively accesses to the associated file's contents. File mappings with special semantics, e.g. a mapping that is backed by anonymous memory, are provided by operating systems via special files that are not represented as physical files.

3.1.4 Communication

POSIX defines the following mechanisms for communication between processes, tailored for notification, synchronization or data exchange.

Signals are *software interrupts* notifying processes of internal or external events, such as program errors, asynchronous I/O events or process control events. The majority of signals can be caught and processed in handler functions provided by the application.

Pipes allow two processes to exchange data in a first-in first-out fashion via the `read()` and `write()` functions. Anonymous pipes connect a parent and a child process, named pipes can be accessed like files by unrelated processes.

Sockets form an advanced and more generalized communication channel between possibly unrelated processes. Sockets support addressing in different name spaces and different protocols for data exchange. Sockets are accessed via the `read()` and `write()` functions. Operating systems usually map their network subsystem functionality to the socket interface.

Message Queues allow one or more processes to write messages, which will be read by one or more reading processes. The operating system is responsible for synchronizing applications that access message queues.

Semaphores provide a synchronization point for processes.

Shared Memory multiple processes can share memory in their virtual address spaces for communication, synchronization and data sharing.

All these mechanisms are protected from processes via POSIX access rights described next.

3.1.5 Security

POSIX introduces the concept of *users* classified into *groups*. They are identified by unique user IDs and group IDs respectively. All files and processes are associated with a user ID and one or more group IDs. Access control is performed by comparing these IDs when processes try to access files or interact with other processes. Note that this protection scheme applies to all objects that are represented as files, e.g. devices.

3.2 The GNU C Library

The GNU C-Library provides such common operations as input/output, dynamic memory allocation or string manipulation as well as a POSIX interface implementation to applications. It is based on a low-level interface consisting of POSIX core functionality and other basic features (e.g. for synchronization).

GLIBC's exposes an architecture-dependent interface which needs to be implemented on new architectures. Stubs are provided for all functions that are part of this low-level interface. Architecture-dependent implementations of the functions are grouped per architecture. Together with the build mechanisms, this increases GLIBC's portability to new UNIX-like platforms.

The build process takes the base architecture as an argument and links the library with the corresponding function implementations. If

such an implementation is not available for the specified platform, the corresponding generic stub is used. This mechanism allows to build the library with a narrow interface to the base platform while applications can access all GLIBC features this interface is sufficient for.

3.3 L4

On the L4 micro kernel, a task represents a set of threads running in an address space; tasks are structured flatly.

The threads in different tasks can communicate with each other via IPC. L4 IPC is synchronous, blocking and unbuffered.

The hierarchical L4 address space concept follows a map-grant model based on user-level pagers.

For more detailed descriptions, see [Lie99, Lie98].

3.4 SawMill

SawMill represents a configurable multi-server operating system framework. It aims at de-composing complex systems into sets of components with well-defined interfaces and well-understood protection mechanisms ([GJP+00]). The basic framework consists of and utilizes a number of system services implemented by libraries and system components which are covered in this chapter.

3.4.1 Dataspaces

The virtual memory model of SawMill is based on the notion of *dataspaces* ([ALP+01]). A dataspace is a logical unstructured data container and can be used to abstract files, frame buffers, anonymous or physical memory and so forth.

Applications can access dataspaces via virtual memory by *attaching* them to a *region* of their address space. Figure 1 illustrates a scenario where dataspaces D_1 and D_2 are rooted in address space A_0 and attached to regions in the address spaces A_1 and A_2. D_2 is additionally attached to a region in A_2.

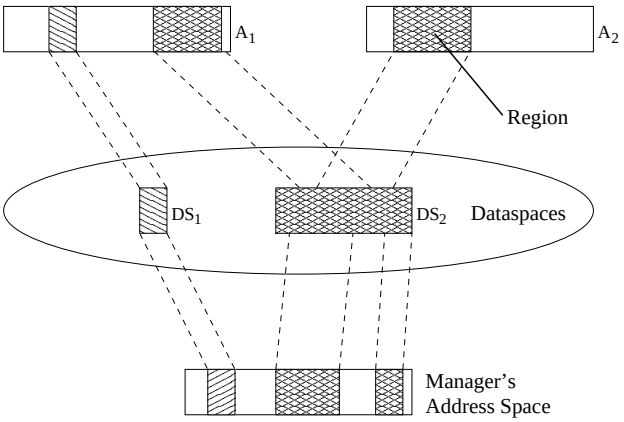

Figure 1: Relationship between address spaces, dataspaces, and regions

Dataspaces are implemented by *dataspace managers* which determine the semantics of their hosted dataspaces. In Figure 1, the application running in address space A0 might represent a dataspace manager.

The *region map* is a address space specific object that manages the association of virtual addresses with regions and the attached dataspaces. It is implemented by a *region mapper* thread, usually running in the address space the managed region map describes. Region mappers act as L4 pagers for other threads running in the address space. When such a thread causes a page fault, the region mapper receives a page fault IPC and translates it into a map request. The request is sent to the dataspace manager associated with the region that covers the faulting address. Both the region mapper and the dataspace manager are free to apply any policy for page fault resolving. A standard implementation of the region mapper is provided by the SawMill environment.

Dataspace managers must support a standard set of operations. Except for *open*, the operations are accessible through a common interface.

Open Opens a dataspace for access. The request provides name resolution (e.g. for files) and can support open modes such as read-only and read-write. After calling Open, a client is authorized to access the dataspace identified by the returned ID.

Close Makes the dataspace inaccessible. All virtual memory pages mapped to the application are revoked. The dataspace ID can optionally be invalidated.

Transfer The authority over the dataspace is moved to another client. The new owner can perform any operation on the dataspace as if it had opened it. The former owner loses any authority over the dataspace as if it hat closed it.

Share The authority over the dataspace is shared with another client. The new owner can perform any operation on the dataspace as if it had opened it. The dataspace is closed after all share-holders have invoked the close operation.

Map Requests a mapping from the dataspace manager at an offset in the dataspace. This operation is used to resolve page faults.

Currently, existing SawMill servers do not fully comply with this standard due to ad-hoc implementations. Also note, that the described operations reflect the semantics of the current implementation and differ from [ALP+01].

3.4.2 Threads and Tasks

SawMill provides a flat view onto tasks, parent-child relationships are not explicitly maintained. Applications can freely create new threads in their address spaces without the interference of any SawMill system entity. L4 allows only a privileged component to create and delete tasks. Conceptually, this role is played by the SawMill task server.

3.4.3 System Services

Resource Management L4 restricts certain operations to a privileged thread which is the *resource manager* in SawMill. It provides access to interrupts, physical memory and task creation and deletion functionality.

Task Management The SawMill task server exports task management functionality to the rest of the system.

Name Resolution The root name server provides a hierarchical name space for the system. Names can be associated with a tupel `[server,handle]` identifying an object in a server. The name resolves a given name either to such a tupel or to a server ID and a remaining name that has to be further resolved by the indicated server.

Anonymous Memory A dedicated dataspace manager provides anonymous memory as dataspaces.

Copy on Write Facility Copy on write functionality can be achieved with a library or with a specific dataspace manager. See the Design section for more details.

Region Mapping Each address space contains a standard region mapper resolving page faults. The region mappers themselves are paged by a central region mapper component.

File Access Files can be accessed as dataspaces at the file provider.

4 Design

This chapter explains the basic concept of our POSIX emulation, the fundamental problems caused by that concept and the strategies employed to reproduce specific POSIX functionality on SawMill.

4.1 The Emulation Approach

Using GLIBC, there are two options to achieve a POSIX emulation on top of SawMill - an additional support layer between SawMill and GLIBC and directly integrating the emulation in GLIBC:

The first alternative requires building a component that offers the low-level functionality required by GLIBC. The C-library itself only needs to be linked directly with that component. It integrates all necessary features (like the POSIX read/write file interface) centrally. This component relies on SawMill system services e.g. to provide file access. All low-level features necessary for GLIBC that do not correspond to SawMill abstractions are handled inside this component.

In the second approach, the functionality required by GLIBC's low-level interface is implemented directly in the corresponding function stubs of GLIBC. Thus, the layer of POSIX compatibility is established inside and on top of GLIBC but not below.

Disadvantages of this solution:

- POSIX compatibility is only given after GLIBC is initialized. During the start-up phase of a process the implementation must take care of this fact.

Advantages of this solution:

- Due to the library concept, the emulation executes inside the client application's address space; therefore it can operate on native SawMill objects (e.g. dataspaces) yielding better flexibility and performance.
- A central component providing POSIX semantics would penalize applications with the IPC overhead caused by communication between the application and that component.
- A component for POSIX emulation represents both a performance bottleneck and a single point of failure.

4.1.1 Decentralization of Data

When moving from a monolithic to a multi-server system, formerly centrally handled data is distributed among components. In our scenario, data is kept inside the process's address space instead of being administered in a central component. This has implications on how the data is accessed, shared and managed.

The removal of a protection boundary between the application and its related data improves access time as no IPC or mode switch is necessary. Sharing of this data between multiple tasks becomes more complex. Tasks either have to arrange for an adequate sharing scheme based on the available system primitives (e.g. shared memory on L4) or they use a central component. Whether data is handled in a central managing component or not depends on many factors, the most important being security and performance.

For example, in our GLIBC emulation file descriptors identify dataspaces or devices managed by system components. Since these components perform the necessary access control operations based on the native SawMill objects, the file descriptor data can be safely handled in the client's address space.

4.1.2 Atomicity

Many POSIX functions need to be performed atomically. If such a function is performed as a system call in a monolithic system, this atomicity is implicitly given (on single-processor machines). Our emulation approach involves the multiplexing of POSIX functions to possibly multiple accesses to SawMill system servers and the handling of data in user space instead of kernel space, thus the atomicity of such functions is generally lost. The implementation has to re-establish pseudo-atomicity of emulated functions through adequate locking and synchronization protocols to retain their semantics.

As an example, POSIX demands that two semi-concurrent invocations of the `write()` function on the same file contents are executed atomically and the file completely reflects the modifications incurred by one of the `write()` calls. When a monolithic system implements `write()` in a blocking fashion this property is implicitely given. With a non-blocking `write()`, it has to be ensured explicitely by the kernel which has full control over the operation.

In our emulation, the `write()` function performs an in-memory copy of data with no system servers being involved (except on page faults). Thus the atomicity property of write needs to be achieved explicitely within the `write()` function executing inside the caller's address space. How atomicity is established in this case is a design issue of the file operations.

4.2 Focussing our Goal

The GNU C library is by now very rich in its concepts and its functionality reaching from extensions like thread packages to application programs. It is clearly beyond the scope of this thesis to provide propositions or implementations for an emulation support of all GLIBC features. Therefore, we have to narrow our focus on a subset of its functionality which supplies a reasonable base of features for applications and keeps the involved implementation effort in a manageable range.

In order to reach a very basic level of functionality the following GLIBC mechanisms were obviously necessary: supporting the POSIX memory model and basic file and terminal I/O. This feature set will get a simple 'Hello World' program up and running.

To provide the essential POSIX interface, the emulation also needs to include process management, process creation via the `fork()` function, signals, advanced I/O functionality like the `select()` function and terminal related functions.

This set now allows running simple POSIX programs and serves as a fundament for further development to enhance and enrich the functionality offered by the GLIBC port.

To illustrate the feasibility of this task, we decided to flesh out the emulation to the extent where a POSIX shell program can be linked and run with our modified GLIBC, allowing users to use the shell for starting other applications linked with the library.

Dynamic linking and loading is also interesting for us as we want to learn about the effects of code sharing on native SawMill programs. Since this functionality can be achieved with an acceptable amount of extra work based on the features mentioned before, we decided to support it both for standard programs linked with the modified GLIBC and for SawMill components.

4.3 Emulation Strategies

The following chapter describe which strategies were chosen to emulate specific aspects and features of GLIBC based on our underlying emulation approach.

4.3.1 Processes

To protect POSIX processes from each other, each process is emulated by an L4 task. Thus, existing SawMill concepts and components can be used easily (e.g. the management of dataspaces via a region mapper thread).

The association between POSIX process IDs and L4 thread IDs needs to be managed within the trusted computing base and be accessible by system servers to prevent client tasks from abusing their POSIX identity. A possible solution would be a process server keeping track of this information as well as providing access to other data related to a POSIX process like user and group ID. For more details see the chapter on future work. The remaining part of this chapter assumes that a service providing this information is available.

4.3.2 Files

The emulation provides access to files via the dataspace paradigm. A file dataspace is mapped into the application's address space for reading and writing. This approach is natural as it gives us the full flexibility of the SawMill dataspace concept allowing, for example, a conceptually very simple integration of copy-on-write semantics or partial mappings of large files.

In the current implementation, the file provider does not integrate with SawMill's name resolution scheme. It uses a separate name space for the files it manages. A generalized approach should initialize the application with a path prefix identifying the root of the application's file name space in SawMill's global name space.

4.3.3 File Operations

The association of a memory region with a file dataspace and the corresponding dataspace manager is established by the `open()` function. Access control is performed on the server side as part this operation.

The `read()` and `write()` operations are emulated by copying memory between the user buffer and the appropriate memory location where the file is mapped. POSIX has very specific requirements about how these functions handle blocking and non-blocking I/O, signals and what error conditions can occur. Such aspects are currently not covered by our implementation.

The `mmap()` operation closely resembles the attaching of a dataspace. All mappings are located in a memory region reserved for this purpose. Requests for mappings of large files can be satisfied by mapping a smaller

dataspace window whose offset is adjusted according to the accessed file locations. If the application requests a private mapping, the emulation uses SawMill's copy-on-write service to establish private copy-on-write access to the file dataspace. If an anonymous mapping is requested, the emulation backs the mapping with an anonymous memory dataspace.

File operations depend on the file they are invoked on. E. g. file access is performed as described above while terminal I/O might be based on IPC communication with a server component. Thus, every file has to be associated internally with its appropriate set of access function which are used for demultiplexing when the generic version of these functions is called by the application or GLIBC.

4.3.4 Signals

An implementation of POSIX signals must provide these fundamental properties:

Signal Sources Signals can be generated *synchronously*, i. e. by the receiver itself, or *asynchronously*, by events outside the control of the receiving process.

Signal Handling Signals cause either a registered handler function to run inside the process's address space or a default action to be performed. Via a *signal mask*, a process can block or ignore specific signals.

Transparency Signal handling is transparent to the process as long as the handler function or default action do not modify its state explicitly.

Nesting Signal handler functions can be interrupted by signals in a hierarchical nesting scheme.

These requirements are met by emulating inter-task signals via IPC. A *signal thread*, which is introduced into every task, is responsible for receiving IPC messages representing signals. When this happens, the signal thread puts the main thread into a generic signal handler function. That function is called directly when the process raises a signal to itself. This allows for signal nesting, it is transparent to the main thread and signal handler functions can be easily integrated.

The POSIX standard demands that a user program may not install handler functions for certain signals. Our emulation keeps the signal handler information completely under the control of user programs, so while the emulation library functions do not permit to establish prohibited user signal handlers, the user program is able to modify this data to circumvent such restrictions.

Access control is performed on the receiver's side in the signal thread to prevent unauthorized processes from manipulating other processes via signals. Signals that non-cooperatively delete a task need to be transformed into a request to the task server. This scheme is to be regarded as a temporary approach as it makes the signal thread vulnerable to denial of service attacks. Instead, signals could for example always be delivered via a signal or notification component.

Error signals caused by hardware exceptions can be emulated via adequate exception handlers installed in the task's interrupt descriptor table (which is in turn emulated by L4).

4.3.5 Copy-on-Write

The copy-on-write technique allows reducing memory consumption and copying overhead for memory copies on systems with shared memory.

Currently, there is a copy-on-write library available in SawMill. It simplifies the implementation of this mechanism for dataspaces in dataspace managers. With a library, copy-on-write versions of dataspaces can only be constructed if the corresponding dataspace manager implements copy-on-write.

A different approach is to construct a copy-on-write dataspace from the source dataspace with a copy-on-write dataspace manager. The resulting dataspace is hosted by that dataspace manager and requires no modifications in the source dataspace manager. In this solution the copy-on-write handling is not optimized for each dataspace type. Its flexibility though was our main motivation to implement such a component.

4.3.6 Loading and Linking

In the SawMill environment loading and linking can be performed either internally or externally.

External loading makes use of a loader component which provides the binary to be run as one or more dataspaces. After attaching them to a new task it starts executing. Dynamic linking can also be carried out by the loader component. Interpreters that may be specified in an elf binary to prepare or undertake its execution cannot be run by the loader component without additional security measures.

With internal loading, a new task starts executing a simple start binary which loads and initializes the binary to be run inside the new task's address space. This process can also include dynamic linking and the execution of elf interpreters.

In our scenario we can specifically benefit from the second approach: with the basic GLIBC features available, existing elf interpreters can be used, namely ld.so, the dynamic linker included in GLIBC.

4.4 Required Base Services

The implementation of the emulation strategies mentioned requires a set of base services on which it can be founded. These services are:

1. name resolution: to dynamically access system components.
2. task management: for creating and deleting tasks.
3. anonymous memory dataspaces: to back memory for management data and application memory requirements like the stack and the heap.
4. copy-on-write for dataspaces: to provide copy-on-write semantics where required by POSIX and to permit access control on dataspaces for copy-on-write data.
5. external region mapper: to flexibly back internal region mappers as part of address space initialization.
6. file dataspaces: to run executables and make files available to the application.

These services are provided by generic SawMill components (see Appendix A for further details on system setup).

5 Implementation

5.1 Files

By design files are emulated via SawMill dataspaces. The SawMill *file provider* component supports resolution of file names and exports file contents as dataspaces[1]. Advanced file system functionality like directory handling is currently not supported. Such features require a full implementation of a file system under SawMill.

File Related Data The files opened by a process are represented by file objects. They are created in the open call and describe the dataspace they are associated with. A file object holds a set of pointers to the operations which can be performed on that file. The state associated with a file like the current read position or open time flags are also stored in the file object structure.

The files structure describes all files currently opened by a process. It contains the process's file descriptor tables and synchronization variables which are used by all functions accessing the file management data to prevent data corruption.

The operations which can be performed on an object identified by a file descriptor depend on the object type and are associated with each file object. In Linux they are retrieved from the underlying inode objects which are currently not available in our emulation scheme. Therefore, the file descriptors 0, 1 and 2 are initially associated with terminal operation functions, all others file descriptors are associated with file dataspace operations on open.

File Operations The `open` operation is responsible for opening and attaching a file dataspace for the specified file, allocating a new file object, assigning it a new file descriptor and augmenting it with all file handling data. These are the associated dataspace id and manager, the position of the dataspace in the address space and all data necessary for implementing the access operations. The `close` operation closes and detaches the file dataspace and frees the management data structures.

The `read` and `write` operations retrieve the file object identified by the specified file descriptor, look up the memory location of the associated file dataspace and copy memory to/from the specified user buffer at the current file position.

Calling `mmap` results in opening the specified file dataspace and attaching it. Each address space contains a pool for such attached dataspaces. Attaching smaller parts of large files is not yet implemented. The `munmap` operation is mostly equivalent to the `close` operation.

If a private mmapping is requested the emulation opens and attaches a copy-on-write version of the file dataspace via the copy-on-write server. For anonymous memory mmappings, instead of a file dataspace a anonymous memory dataspace from the `dm_anon` SawMill server is used. Read or write access control to mmapped memory is currently not supported.

[1] Although file dataspaces are writable, the modified data is not written back to disk by the file provider, i. e. writing to a file is not supported.

5.2 Memory Management

Application Heap The application's heap is managed by GLIBC's `malloc()` and `free()` functions. These functions require the functions `brk()`, `sbrk()` and `getpagesize()` to be implemented.

The `getpagesize()` function referenced by `malloc()` currently returns a fixed value of 4096 bytes representing the default page size on the x86 architecture.

The `brk()` function, responsible for resizing the data segment, is emulated by adjusting a break value in a anonymous memory dataspace. Currently, a fixed size area in the task's address space is reserved for this dataspace. The break value can range between its lower and its upper end as requested by GLIBC. This scheme can be improved by extending this range dynamically to keep the reserved but unused space in the address space and region map respectively small.

The `sbrk()` function, which is equivalent to `brk()` except using a relative instead of an absolute value, wraps `brk()` in a straightforward manner.

Application Stack The application's stack currently resides in a fixed memory range and is backed by an anonymous memory dataspace. This solution is sufficient for the moment but needs to be improved by stack overflow detection and handling.

5.3 Signals

Signals are emulated via IPC received by a signal thread which is run in each task. When a signal IPC arrives, the signal thread puts the main thread into a signal handler function. The previous execution context as well as the signal ID make up the arguments of this handler function which, after handling the signal, resumes to the code executed when the signal arrived.

The signal handling function is responsible for managing signals in a POSIX compatible fashion. It checks whether signals are ignored or blocked or whether the application specified handler functions to be executed. It also checks for pending signals (i. e. signals which were blocked when they arrived and were unblocked later).

Signal-Related POSIX Functions

kill() Identifies the signal thread of the target process and sends a signal IPC without blocking.

pause() Waits for an IPC from the nil thread effectively blocking the main thread until the signal thread calls `l4_thread_ex_regs()`. If a signal was handled successfully, `pause()` returns, else it blocks again.

sigsuspend() Works like `pause` except the signal mask is modified as requested.

raise() Calls the signal handling function with the specified signal ID.

sigaction(), sigalstack(), signal(), sigprocmask(), sigstack() Modify the signal handling data (signal masks, registered handlers etc.).

5.4 The Copy-on-Write Server

Bull, the system server providing copy-on-write semantics for dataspaces, is implemented as a dataspace manager. Clients can open a copy-on-write copy of any existing source dataspace by providing bull with this source dataspace and an anonymous memory dataspace. Bull then provides the client with pages from the source dataspace as long as the client only reads from these dataspace portions. On a write access, the contents of the source page are copied to a page from the anonymous memory which is handed to the client.

6 Conclusions

Based on decision to use the GNU C library for a POSIX emulation on top of the SawMill multi-server operating system we investigated and proposed strategies for implementing specific aspects of the emulation. Also an implementation is provided that allows linking and running simple applications with the modified library. The goal of a functional shell application in the emulation environment has not been reached mainly due to the underestimated implementational effort.

We marked out additional requirements for the SawMill dataspace framework that we will bring in to the discussion about a generic L4 environment specification. Other new frameworks and services were identified from which the emulation library as well as future projects would benefit.

The implementation can serve as a basis for evaluating the interfaces of SawMill components using POSIX compatible applications. As a platform for performance tests and feasibility studies it can aid the design and development of future components or services in SawMill.

7 Future Work

7.1 The fork() and exec() Functions

The duplication of address spaces is required for emulating the `fork()` function. Using a copy-on-write strategy to improve performance is a must. An address space duplicate in the SawMill environment can be achieved in two ways: either the parent creates copies of all its dataspaces and installs a patched region map in the child or it installs a copy of its region map in the child and grants it access to all its dataspaces.

The first approach is intransparent and demands that all occurrences of dataspace IDs in the parents address space are changed in the child to the IDs of the dataspace copies. Thus the emulation has to manage all dataspace ID related data centrally.

The second approach is transparent, the parent's dataspace IDs remain valid in the child. After a parent shares its dataspaces with the child, the dataspace managers have to provide copy-on-write semantics to the two clients on these dataspaces.

In the current implementation, copy-on-write dataspace copies can become inconsistent if the source dataspace is modified after the copy is created. Inconsistencies can be prevented by introducing copy-on-write semantics on the source dataspace, too. Depending on the address space duplication scheme, dataspace managers additionally need to do this transparently.

With the `exec()` function a process discards its current execution context and requests to load and run a specified program. While our current tools can be used for loading and linking the new program, freeing all resources, mainly dataspaces, no longer accessed by the process has to be performed in a systematic and reliable manner. What this procedure looks like in detail needs to be revealed by further investigation.

7.2 The select() Function

The POSIX function `select()` serves for blocking on a set of file descriptors until activity is detected on one of them or until a timeout occurs. The multi-server approach of SawMill presents a fundamental problem when emulating such a functionality. Since the specified file descriptors might identify objects in a large number of system components, the event notification scheme possibly involves a large amount of communication. Thus, an approach needs to be devised that optimizes the number of IPCs for registration and notification, scaling well with the amount of clients, servers and invocations of the `select()` function. Results in this area could then contribute e.g. to a generic event notification framework for SawMill.

7.3 Resource Management

A framework for resource management in SawMill would be desirable, not only for POSIX emulation but also for other scenarios where excessive usage of resource has to be prevented.

SawMill's component structure particularly demands scalability for systems with large amounts of servers and clients from such a framework but it also has to regard many other aspects like resource representation, allocation or control schemes and security.

Reclaiming Resources One aspect of resource control important for task management in SawMill is the non-cooperative de-allocation of dataspaces after a task stops executing. It needs to be robust against broken or malicious tasks and scale well with the number of involved dataspace managers and clients.

One possible solution centers around a component in the operating system's trusted computing base responsible for task creation and deletion which is the task server in SawMill. From this component dataspace managers learn of a task's death and de-allocate resources associated with dataspaces still held by the dead task. They can either receive death notifications about dead clients from the task server or poll it for this information.

Polling Depending on the implementation, the IPC load grows with the number of clients, of dataspace managers and with a reduction of the polling interval. The polling interval is a major weakness of this concept because a fair trade-off between vulnerability in the form of unnecessary resource allocation and excessive IPC usage has to be found.

Death Notifications Death notifications are sent by the task server to all dataspace managers that registered with it. Dataspace managers can either register once with the task server and receive messages on any client's death or register and de-register for each client trading registration IPCs against notification IPCs.

Which scheme is appropriate under which circumstances has to be determined by an in-depth evaluation.

7.4 Process Management

Emulated processes and system servers require information about other processes e.g. for communication or authentication. Such a lookup scheme has at least to export the mapping from POSIX process IDs to L4 thread IDs and vice versa, the persona of a process, i.e. the user and group impersonated by the process, and parent-child relationships. One possible solution would be to collect this data in a central system component from which it can be retrieved.

7.5 Terminal I/O

An emulation of terminal input and output still needs to be integrated into the POSIX emulation. It could be based on existing SawMill emulations of a character device or the I/O server. The implementation would be simplified since the interfaces of these components are POSIX oriented. Yet a systematic investigation should reveal whether such an interface is efficient and flexible in the SawMill environment.

7.6 Additional POSIX Features

The potential and usefulness of the limited POSIX emulation created in the course of this project is obviously increased by adding more emulated features. Existing SawMill components like a network stack or a file system are one set of possible choices. New functionality could be added both to SawMill and the emulation e.g. in the form of a thread package.

The motivation for such efforts is not only the support of a larger application base by the emulation. Again, new insights on component interface design and its impact on application performance are expected.

7.7 Performance Evaluation

Performance tests of the modified GLIBC and the involved SawMill components in part motivated this project but are yet outstanding due to the unfinished implementation status.

Based on performance numbers we can judge the presented design decisions, identify flawed concepts and get hints about superior alternatives. Comparisons against L4Linux can reveal architectural costs imposed by our design and indicate its competitiveness. Of particular interest might be how the overall application performance is affected by the emulation.

8 Appendix A: The Start-up Environment

In our design we decided for internal loading of applications. This section describes which concepts and components are involved when starting an application from scratch, i. e. in the standard SawMill environment and not via a combination of the emulated `fork()` and `exec()` functions.

The Start-up Components The following components implement our loading and linking scheme:

`runl4` serves as a front end to the loading library and allows to run applications from L4Linux.

The loading library is responsible for starting the startup binary in a new task.

The startup binary loads and relocates the application to be run. It also contains the internal region mapper.

The external region mapper is the L4 pager of the new task's internal region mapper.

`ld.so` is GLIBC's dynamic linker and is run by the startup binary if specified as an interpreter in the application's elf image.

The Start-up Procedure

1. `runl4` opens a file dataspace for the startup binary and passes it to the loader library along with the IDs of system server to be user by the new task.
2. The loader library parses the elf image of the startup binary and pokes the system server IDs and the name of the application to be run into the environment section of the startup binary. It registers the file dataspace of the startup binary with the external region mapper and starts the startup binary in a new task with the external region mapper as its pager.
3. The internal region mapper included in the startup binary starts and pages the main thread with the loading code.
4. The loader code reads the data contained in the environment section. It opens a file dataspace of the application to be executed, scans the elf image and relocates it. Then it puts environment information (system server IDs, program arguments, environment strings etc.) on the application's stack and runs the application code or an interpreter if specified in the elf image.
5. `ld.so`, the interpreter of dynamically linked applications from GLIBC, performs the necessary runtime linking and finally starts the application code.

9 Appendix B: Installation and Usage

9.1 Installing SawMill

The installation of SawMill is described at http://i30www1.ira.uka.de/ sawmill/.

Before building SawMill, make sure you use the new loading scheme of SawMill by updating the makefiles after checking out the sawmill CVS module: `cvs update -r RM2 sawmill/sawmill/Makeconf sawmill/sawmill/pkg/Makefile sawmill/sawmill/pkg/Makefile.tmpl`.

9.2 Installing the GLIBC

To build the SawMill version of GLIBC follow these steps:

1. check out the CVS module `glibc-2.2`
2. create a `build` and an `install` directory
3. change into the `build` directory
4. run the configure script: `../glibc-2.2/configure --host=i386-sawmill --enable-hacker-mode --disable-sanity-checks --disable-profile --prefix=<installation directory (e.g. /home/foo/src/glibcinst)> --with-sawmill=<sawmill directory (e.g. /home/foo/src/sawmill)>`
5. run `make` and `make install`

During the build process you might encounter the following problems:

- If the build process aborts complaining about a missing `sln` program you can generate it manually: `gcc /src/glibc-2.2/elf/sln.c -o elf/sln`. Then restart the build process.
- If the build process aborts complaining about a missing `iconv_prog` program just do a `touch iconv/inconv_prog`.

9.3 Compiling and Linking Applications

To compile an application with the SawMill GLIBC invoke gcc so that it uses the include directory in the GLIBC installation directory and the include files of your gcc version instead of the standard include files. In our setup this looks like: `gcc -Wall -c -nostdinc -I /glibcinst/include -I/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include start.S <source files>`.

Linking the application with the modified GLIBC needs the following options:

- library paths: the lib directories in the GLIBC and SawMill installation directories as well as your compiler libraries
- object files: start.o and crti.o providing the .ini elf section and crtn.o providing the .fini elf section, all of which can be found in the ¡glibc build dir¿/csu/ directory plus your application object files
- libraries: c, gcc, thread, semaphore, l4util, rm2, rm2-server, utils, genericdm

In our setup the ld invocation looks like

```
ld -Bdynamic --dynamic-linker=~/glibcinst/lib/ld.so.1 \
 -L~/gsm/glibcinst/lib -L~/sawmill/sawmill/lib \
 -L/home/cross/gcc-2.95.2/lib/gcc-lib/i686-pc-linux-gnu/2.95.2 \
 start.o ~/glibcinst/lib/crti.o ~/glibcinst/lib/crtn.o \
```

```
<application object files> \
-lc -lthread -lsemaphore -ll4util --start-group -lrm2-server \
-lrm2 --end-group -lutils -lgenericdm -lc -lgcc \
--rpath=/home/sgoetz/gsm/glibcinst/lib \
--rpath=/home/sgoetz/gsm/sawmill/sawmill/lib
```

References

[ALP+01] Mohit Aron, Jochen Liedtke, Yoonho Park, Luke Deller, Kevin Elphinstone, and Trent Jaeger. The SawMill framework for virtual memory diversity. In *Australasian Computer Systems Architecture Conference*, Gold Coast, Australia, January 2001. IEEE Computer Society Press.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Symposium on Operating Systems Principles*, pages 120–133, 1993.

[GJP+00] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[Kor97] David G. Korn. Porting UNIX to Windows NT. In *1997 Annual Technical Conference*, pages 43–57. USENIX, January 1997.

[Lie95] Jochen Liedtke. On micro-kernel construction. In *Symposium on Operating Systems Principles*, pages 237–250, 1995.

[Lie98] Jochen Liedtke. *Lava Nucleus (LN) Reference Manual*. IBM T. J. Watson Research Center, 2.2 edition, March 1998.

[Lie99] Jochen Liedtke. *L4 Nucleus Version X Reference Manual*. Universität Karlsruhe, x.0 edition, September 1999.

[Noe98] Geoffrey J. Noer. Cygwin32: A free Win32 porting layer for UNIX applications. In *2nd USENIX Windows NT Symposium*, page 31. USENIX, August 1998.

[Wal97] Stephen R. Walli. OPENNT: UNIX application portability to Windows NT via an alternative environment subsystem. In *USENIX Windows NT Symposium*. USENIX, August 1997.