Loading and Debugging Tasks in Sawmill

A Dissertation submitted to the School of Computer Science and Engineering of The University of New South Wales in partial fulfillment of the requirements for the degree of Bachelor of Engineering (Computer Engineering)

Luke Deller (2172287)
Supervisor: A/Prof Gernot Heiser
Assessor: Dr Yoonho Park

October 1999

Abstract

Sawmill is a highly configurable multi-server operating system built on the Lava nucleus, currently under development at IBM's T. J. Watson Research Center. This thesis describes some contributions that I made to the project during my stay at Watson.

Preparing a task for execution in a multi-server environment is not as simple as in a monolithic kernel because a number of different components need to co-operate to provide the necessary services to start the task. A general design for loading tasks in Sawmill is presented and its rationale explained. My implementation of a system for loading tasks from ELF files is described and issues which arose have been discussed.

Debugging an experimental operating system and its processes is traditionally quite tedious, often featuring serial port debugging output and frequent reboots. Interactive loading of tasks removes much of the need for rebooting, but complex debugging can still be difficult. I have ported a popular symbolic debugger to the Sawmill system, which allows multi-threaded Sawmill tasks to be debugged. Some implementation issues encountered in debugging on the Lava nucleus are discussed.

The Sawmill system aims to provide Linux binary compatibility which involves supporting the services which are usually provided by the Linux kernel. A strategy for migrating virtual memory support from the Linux kernel to Sawmill services is presented.

Acknowledgements

I am greatly indebted to all the members of the Sawmill group for their kindness and support during my visit with them: Yoonho Park, Trent Jaeger, Kevin Elphinstone, Alain Geflaut, Jochen Liedtke, Volkmar Uhlig, Chris Howsen and our manager Richard Neves. It was a pleasure to be with a group which worked so closely together — one of the highlights of the work there was the frequent group discussions, and the chance to talk about issues with other Sawmill people. I am particularly grateful to Yoonho Park for letting me invade his office and share his time.

Contents

1 Background **1**
1.1 Dynamically Configurable Operating Systems 1
1.2 Multi-server Operating Systems 1
1.3 Sawmill 2
1.3.1 Goals 2
1.3.2 Structure and Development 2
1.3.3 Services 4

2 Loading a Task **5**
2.1 The Sawmill VM Framework 5
2.1.1 The Model 6
2.1.2 Implementing on the Lava nucleus 7
2.2 Design of Loading in Sawmill 7
2.2.1 Decoding Executable Files 7
2.2.2 Supporting a Region Mapper 8
2.2.3 Loading and Starting a New Task 8
2.3 Implementation 10
2.3.1 The Region Mapper 10
2.3.2 A Library to Support Copy-on-Write 11
2.3.3 The Elf Decoder 12
2.3.4 The Region Map Manager 14
2.3.5 The Linux File Provider 15
2.3.6 Running a Sawmill Component from Linux 15

3 Debugging a Component **17**
3.1 A New Serial Protocol for GDB 17
3.2 The Sawmill GDB Stub 18
3.3 Debugging Support in the Loader 20
3.4 A Brief User Manual for DDD on Sawmill 21

4 Managing Address Spaces of Linux User Tasks **23**
4.1 Accessing User Memory for Handling System Calls 23
4.1.1 Autonomy of User Tasks 23
4.1.2 Controlling External Data Accesses 24
4.2 Migration Strategy 25
4.2.1 What Needs to be Changed 25
4.2.2 Small Steps 26

5 Conclusions **28**

List of Figures

1.1 L4/Linux. 3
1.2 Outline of the ultimate structure of Sawmill 3
1.3 Developing Sawmill 4

2.1 Loading and starting a task. 9
2.2 The dataspace hierarchy in my implementation of the ELF decoder. 13
2.3 Removing a level of indirection from the dataspace manager hierarchy. . . . 13
2.4 The Region Map Manager 15

Chapter 1

Background

1.1 Dynamically Configurable Operating Systems

The benefits of having dynamically configurable operating systems seems to be widely accepted. Many systems support the insertion of modules into the kernel by a privileged user, for example the kernel modules in Linux [4] and FreeBSD [6].

One cause for concern with such systems is that arbitrary code can be inserted into the operating system kernel by a privileged user. Any code that is inserted must be fully trusted, because any bugs or hostile code is unrestrained and can easily violate the system. This is a bad software engineering practice which can lead to a less robust system.

Several research projects are investigating methods of protecting against bad kernel modules. The SPIN operating system [1] relies on restricted dynamic linking enforced by the type safe properties of the Modula-3 language to protect the kernel. The VINO operating system [2] uses software fault isolation for safety and a lightweight transaction system to cope with module resource hoarding.

It is interesting to note that the above Vino paper acknowledges that the performance cost of their safety techniques are high compared to the cost of the extensions themselves, yet they still insist that the benefits brought by the extensibilty are well worth the cost.

1.2 Multi-server Operating Systems

Rather than having a monolithic kernel with all kinds of operating system services thrown together in the same address space, an alternative approach is that of a *multi-server* operating system. In such a system, only a very small kernel, called a *microkernel*, runs with the CPU in privileged mode. This kernel provides only very simple abstractions such as threads, address spaces and inter-process communication (IPC). The rest of the system is composed of distinct components running in their own address space. Often these components provide orthogonal services. One of the pioneering multi-servers is MachUS [3]. Further examples of multi-server systems are Sun's Spring system [8] and ChorusOS [7].

Multiserver operating systems potentially offer high dynamic configurability of the system without the drawbacks of inserting code into the same address space as the rest of the system. They do however introduce new challenges in system design due to the distributed nature of the system.

1.3 Sawmill

1.3.1 Goals

Sawmill is a multi-server operating system which runs on IBM's Lava nucleus, which is a development of the L4 microkernel. Sawmill aims to support the Linux Application Binary Interface (ABI) in addition to any other application interface which may be desired in the future.

One of the major design goals of Sawmill is to have a highly configurable operating system. The Linux kernel already has some degree of static and dynamic configurability, but our approach differs from the existing system in two ways: our componentisation is more aggressive than the current modularisation of Linux; and we can support much more robust security models between components than is possible between Linux modules.

The Lava nucleus upon which Sawmill is based supports only very simple abstractions: threads, address spaces and inter-process communication (IPC).

1.3.2 Structure and Development

The starting point for development of Sawmill was L4/Linux from Dresden. Using the work from Dresden, the version 2.2.1 Linux kernel was ported to run on the version X Lava nucleus. The structure of the L4/Linux system is described in Figure 1.1. The Linux kernel was run as a single Lava task, and each Linux user process was run as another Lava task. The task for each Linux user process was paged by the Linux kernel, and had an "emulation library" mapped into its address space. The emulation library would catch exceptions (including system call traps) in the user task, and would convert the exception into an IPC to the Linux kernel. The emulation library would save all user registers into a memory page which was shared with the Linux kernel. Since the Linux kernel also backed all of the user task's memory, it had complete access to the user's registers and address space. Thus all system call handling was performed in the Linux kernel, with the emulation library merely serving as a "trampoline".

Our vision for the structure of Sawmill is quite different. We aim to have a multi-server style system where different OS services are provided by different Sawmill components, as depicted in Figure 1.2. Consequently we are looking at making the emulation library much more substantial: it needs to decode system calls and translate them into requests to the appropriate Sawmill component. Some system calls could even be processed by the emulation library alone. The Sawmill components will preferably be generic enough to be used for supporting application interfaces other than the Linux interface.

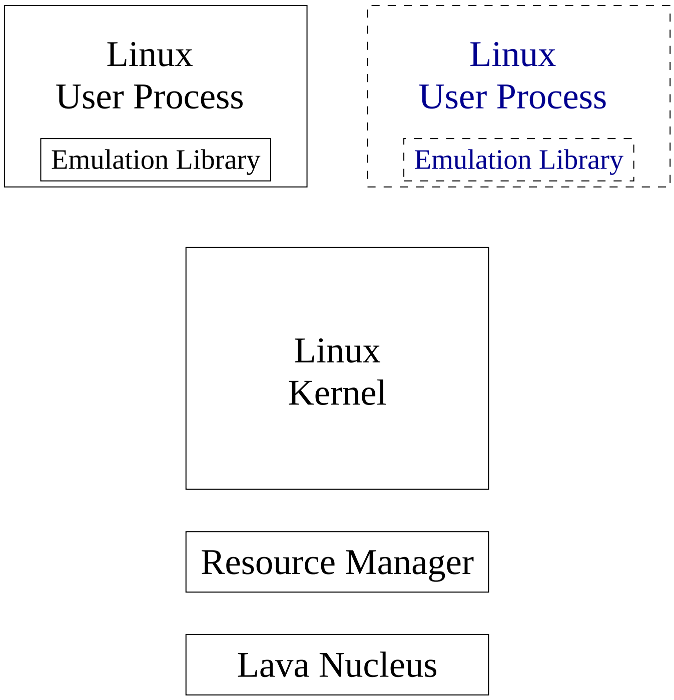

Figure 1.1: L4/Linux.

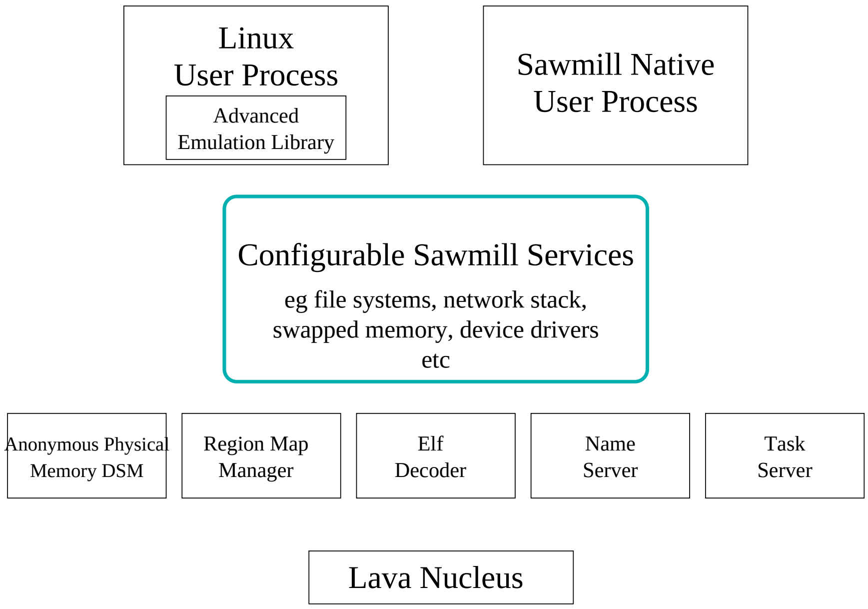

Figure 1.2: Outline of the ultimate structure of Sawmill

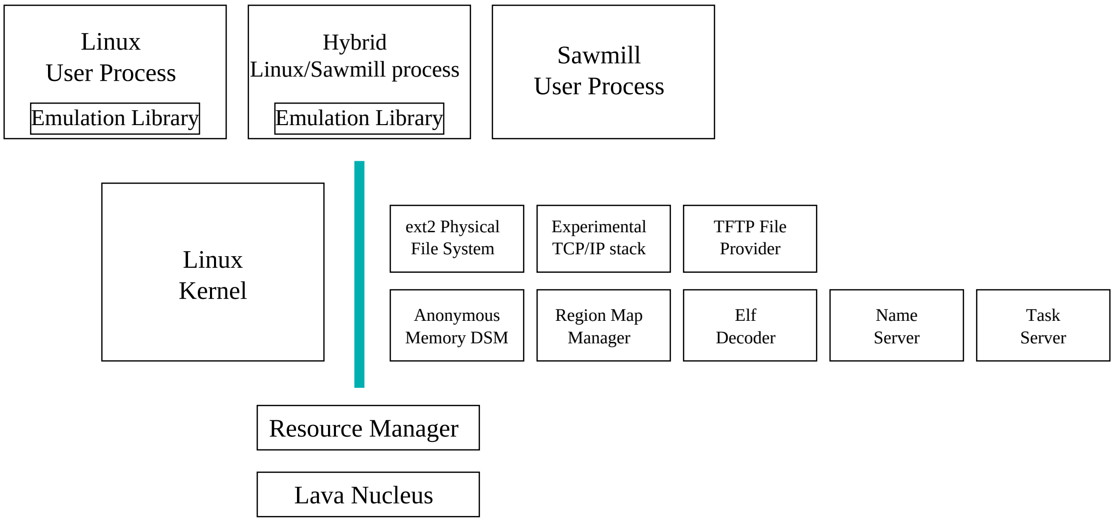

Figure 1.3: Developing Sawmill

We are employing a two-pronged technique to develop such a multi-server from our starting point of L4/Linux. Firstly, we are building basic infrastructure and components from scratch running besides Linux. Secondly, we are separating parts of the Linux kernel code into standalone components. Ultimately we hope to see the Linux kernel task becoming redundant when we can run a complete system outside of it. The structure of our intermediate development system is shown in Figure 1.3.

1.3.3 Services

System functionality in Sawmill is provided by *services* whose interfaces are currently specified in a variant of CORBA's Interface Description Language (IDL) adapted for Lava nucleus IPC.

One Sawmill component could provide several services. Two implementations of a service which use the same interface can coexist; these are regarded as two services.

A nameserver can provide a reference to a service, after which the functions exported by the service can be called. Automatically generated stubs handle the marshalling and IPC necessary to perform the remote procedure call.

Chapter 2

Loading a Task

A new L4 task needs to be started in a Sawmill system when a user process is created (eg when a Linux process forks) or when an OS component is inserted into the system. At the minimum, to create a nucleus task requires two things: an initial address for the instruction pointer (EIP); and another task which can fulfill the first page fault at this initial EIP. This chapter describes the design and implementation of a scheme to load and run Sawmill task within our virtual memory framework, starting from this very primitive mechanism.

My implementation for the various Sawmill loading components was begun from scratch. At that time we were starting from a plain L4/Linux platform. However several advances in Sawmill were built concurrently which affected my work:

- The Lava nucleus was modified to use a different IPC format which allowed more data to be transferred in a "short" IPC (ie involving no long message buffer in memory but just using registers).
- The CORBA IDL based remote procedure call tools were ported to use Lava IPC, thus allowing us to formalise the interfaces between tasks. This required me to reimplement much of my code to comply with the new client/service model.
- A task server was designed and built. It serves as a proxy for the resource manager's task creation and deletion functions, and will eventually replace this aspect of the resource manager's duties.

The design and implementation of the loading components evolved from the initial attempt as Sawmill developed and my understanding of the issues involved grew. I will present my final design and implementation in the rest of this chapter, but first I need to describe the Sawmill virtual memory framework upon which my design is built.

2.1 The Sawmill VM Framework

Supporting the address spaces of Sawmill tasks is founded upon Sawmill's virtual memory framework which was mostly designed before I joined the group [5].

2.1.1 The Model

We use the term “virtual memory”(VM) distinctly from “disk swapping”. The Sawmill VM framework is a framework for manipulating virtual address spaces and backing them by physical memory.

The basic abstraction for memory is a *dataspace*. It is not necessarily associated with any address space (or rather it has its own address space) — given a dataspace offset, there is potentially a page of memory which corresponds to that offset.

A task's address space can be split into non-overlapping page-aligned *regions*, and a dataspace can be *attached* to a region. The region acts like a window into the dataspace; the first page of the region corresponds to a specified page of the dataspace, and subsequent pages of the region correspond to subsequent pages of the dataspace.

Each dataspace is exported by some *dataspace manager* (DSM). The dataspace manager is a service which must export some method to **open** a dataspace and **close** a dataspace. Opening a dataspace might involve creating a new dataspace or looking up an existing dataspace, depending on the semantics of the dataspace. A dataspace manager can also provide methods to:

- **clone** a dataspace;
- **migrate** a dataspace to a different dataspace manager;
- **copy** a dataspace to a different dataspace manager;
- **share** a dataspace with another client task.
- **transfer** a dataspace to another client task.

Additionally a dataspace manager is responsible for backing the dataspace by exporting a dataspace fault method which is called by the region mapper (described later).

We do not specify any restrictions on the following properties of dataspaces, although the semantics of a particular dataspace might force some restrictions:

- unlike a region, a dataspace does not necessarily have a size (besides the maximum size imposed by finite precision dataspace offsets);
- a dataspace is not necessarily pinned in primary storage;
- the pages backing a dataspace might not exist in any level of storage but could be generated on the fly;
- some the pages of a dataspace could be arbitrarily unmapped from the client by the dataspace manager at any time;

A client may or may not be able to write to a dataspace. Even if it is allowed to write to a dataspace, it might choose to map the dataspace read-only to a region.

A Sawmill dataspace is a similar concept to a Mach memory object [11], but it differs in that more advanced operations can be performed upon a dataspace, for example cloning and migrating.

2.1.2 Implementing on the Lava nucleus

In the lava nucleus, each address space has a set of threads which share and are limited to this address space. Each thread has its own *pager* which handles page faults generated by that thread.

To implement our model, we need a pager assigned to each region or dataspace rather than a pager assigned to each thread. This is because different regions can be backed by different dataspace managers, but a pagefault on a region can be handled in the same way regardless of which thread caused the fault.

To support per-dataspace pagers, we need to have a thread which converts from nucleus per-thread pagefaults to "dataspace faults". We call this thread the *region mapper* because it maps from region to dataspace. The region mapper may or may not run in a different address space from the task that it services.

2.2 Design of Loading in Sawmill

2.2.1 Decoding Executable Files

We cannot run a task without mapping some code into its address space. Usually a task needs to have access to some writeable memory too for storing state. Thus we need some description of the contents of the initial address space of the task. I will call such a description an *executable file*. In the Sawmill system, an executable file matches the abstraction of a dataspace quite well, so I decided to assume that an executable file is a specific kind of dataspace.

Common executable file formats (eg ELF, COFF) describe the address space contents by describing a set of *program segments* or *sections*. The executable file specifies the contents and attributes of each section and also can specify where in the address space it should be placed.

The contents of a section can be nicely represented as a dataspace. Specifying the positioning of sections in the address space then is reduced to specifying the attachment of dataspaces to regions. I call such a description a *region mapping*.

So, to load a task in Sawmill, we need a service which will take an executable file and produce a region mapping, making sure that each dataspace in the region mapping matches a section described in the executable file and has the correct contents. I call such a service an *executable decoder*. This service also needs to obtain the program entry point from the executable file, if it has one.

Notice that section dataspaces could be exported by the executable decoder or by a separate dataspace manager. In the latter case, the executable decoder would still need to ensure that the dataspaces have the correct contents. The executable decoder also needs to make sure that the new task's region mapper has permission to open the dataspaces described in the region mapping.

2.2.2 Supporting a Region Mapper

As described above, a region mapper is a thread which converts page faults to dataspace faults. A newly created task needs to have a region mapper installed for it.

The first design decision that needs to be made is whether to put the region mapper in a different address space from the task which it services. Some advantages of putting it in a different address space are:

- it prevents the task from corrupting its region mapper's data structures
- it might be useful for imposing some restrictive policy on the address space usage of a task.

Reasons for putting the region mapper in the same address space as the task which it services are:

- more of the IPC is intra-task which is cheaper on the nucleus;
- using two separate tasks complicates the authorization mechanism for dataspace managers because the task which opens a dataspace is different from the task which sends dataspace faults.

Since we had no need to pay the performance cost of putting the region mapper in a separate address space, it was preferable to put it into the same address space.

Now it must be determined how to install a region mapper into a newly created task. This needs to be done before the program sections in the initial region mapping can be attached to regions in the task's address space.

When a Lava task is created, an external pager for the first thread is specified. This external pager can be used to map the region mapper's code into the task. Once the region mapper is running, all further threads in the task can be paged by the region mapper rather than by an external pager.

I call this external pager a *region map manager*. It needs to be able to page region mappers in several different tasks. I do not expect that there will not be many variant region mapper implementations in a system. Each implementation will be supported by a different region map manager service.

The region map manager must be able to report its region mapper's entry point.

At task startup, the region mapper needs to set up a given initial region mapping and start another thread to begin the program at a given program entry point.

2.2.3 Loading and Starting a New Task

There are several steps involved in loading and starting a new task. The initial loader design included a service called the *createserver* which performed all the steps to get a task up and running, but it became clear that these steps could easily be performed by a library function rather than a dedicated service.

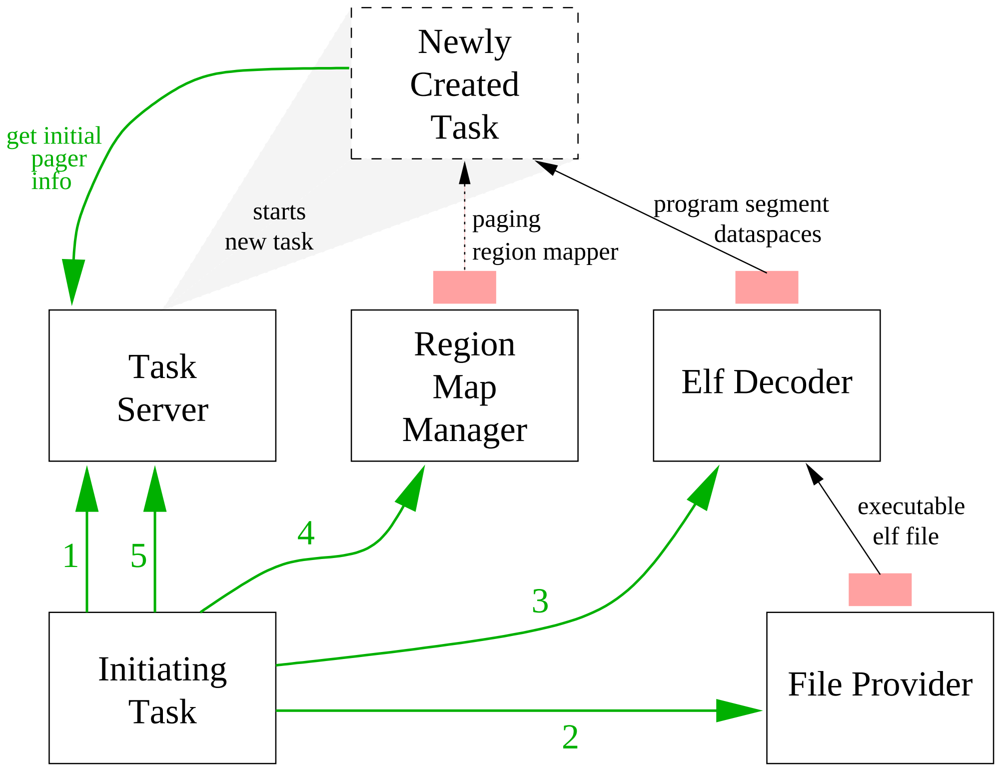

Figure 2.1: Loading and starting a task.

Figure 2.1 shows the steps that are taken to load a task in the implemented system. This is an example of the general procedure required to load and start a new task. The numbers in the diagram correspond to the following general steps that are required to load and start a task are:

1. An executable file dataspace must be obtained somehow (eg from a named file service)

2. A nucleus task number must be reserved for the new task (eg by calling the system's task service)

3. The file must be passed to an appropriate executable decoder (one which understands the executable file's format). The executable decoder will return the initial region mapping and the program entry point.

4. An agreement must be reached with a region map manager to page the region mapper for the new task.

5. The region mapper in the task can now be started as the first thread. Parameters can be passed to the region mapper via the task server, which supports *initial pager information*.

The new task can now initialise itself. Its first page fault will be fulfilled by the region map manager. The region mapper will obtain the initial pager information from the task server, which contains the initial region mapping and the program entry point. The region mapper can then open and attach each entry in the region mapping, and start a thread at the program entry point, paged by the region mapper.

2.3 Implementation

I have implemented several components to comply with this design for loading a task. The current implementation allows a Sawmill component to be loaded from L4/Linux. Several of these components could be used in the future to load Linux user tasks, once the management of user tasks' address spaces is moved out of the Linux kernel (see Chapter 4).

2.3.1 The Region Mapper

The region mapper catches a task's page faults and forwards each one to the appropriate dataspace manager as a dataspace fault. It keeps a track of which dataspaces are attached to which regions. The program can call the region mapper to attach or detach dataspaces. The region mapper which I have implemented runs as the first thread in each task (called lthread 0).

The first implementation of the region mapper receives a page-fault IPC, then IPC-calls the appropriate dataspace manager to obtain the desired page, and finally sends a blank IPC reply to the faulting thread. This has room for improvement: firstly, if the dataspace manager takes a long time to obtain the page then the single-threaded region mapper cannot service other threads for a long time; and secondly, it seems that there is one more IPC than necessary. Indeed the first problem can be solved by eliminating an IPC from the protocol, if somehow the dataspace manager can reply directly to the faulting thread. This frees up the region mapper as soon as the dataspace fault has been sent to the dataspace manager.

However the dataspace manager cannot reply directly to the faulting thread, because the faulting thread is in a closed wait on the region mapper thread. Only the region mapper thread has the right to respond to the faulting thread — it would be nice if somehow the region mapper could pass on this right. It seemed that this problem would be quite common in a system where services make requests on behalf of their clients, so Jochen Liedtke solved it by implementing *IPC propagation* in the Lava nucleus.

There is however one problem with having the dataspace manager reply directly to the faulting thread. The faulting thread is waiting for a reply to its pagefault, with a receive fpage covering its whole address space (ie it will accept any page mapping to any part of its address space). If the mapping from the dataspace manager is sent to the region mapper, then at least the region mapper can restrict its receive fpage.

A solution which solves the problem in the first implementation of the region mapper blocking for too long, and also solves the security problem of the second method described above, is as follows. When the region mapper receives a page fault, it sends a dataspace fault IPC to the appropriate dataspace manager, but using IPC deception making the IPC appear to come from the faulting thread. Then the region mapper `ln_thread_ex_regs` the faulting thread to some code which does a closed IPC receive operation on the dataspace manager with a restricted receive fpage. This piece of code should jump back to the faulting instruction once it receives the IPC from the dataspace manager.

Although this method does not reduce the number of system calls compared to the first method, it does have a significant performance advantage for a multithreaded client with slow dataspace managers because the region mapper thread is not blocked until the dataspace manager has fulfilled the fault; it only blocks until the dataspace manager accepts the fault which should be very fast.

Since we have not encountered in practice the problem with the first implementation of the region mapper, I have not yet fixed the implementation to be as described above, but this would be only a minor modification to the source code.

2.3.2 A Library to Support Copy-on-Write

When I was implementing the ELF decoder and region map manager, there was a common problem that I encountered. Both of these services have multiple clients, and they both map initialised writeable pages to their clients.

There is a blueprint original page in the address space of the service, but when the client writes to the page, the service needs to make a client-specific copy of the page.

Moreover given a client and a blueprint page, the service must be able to find whether that page has been copied for this client yet. If so, it must find where the right copy is; otherwise it must make a copy.

I wrote a copy-on write support library whose interface includes the following function:

```
void *get_page_copy (void *orig_page, unsigned copy_id);
```

This function returns a pointer to a copy of the `orig_page`. Not more than one copy is ever made per copy_id. For storing the copies, the library uses a pool of memory obtained from the system's anonymous memory dataspace manager.

The library also supports deallocating all copies associated with one copy_id, which is needed when a client is lost.

I extended the library to support the provision of pages initialised with zero rather than with a blueprint page. This is useful for the ELF decoder to provide pages for the `bss` section. As well keeping different copies per client, I also needed different copies per client virtual address, because each client might need several different zero-initialised pages.

The problem solved by this library is essentially a multi-client copy-on-write problem. This library was used by both the ELF decoder and the region map manager, and I hope that it can be used again as I am sure that the same problem will arise again.

2.3.3 The Elf Decoder

Given an executable file in the ELF format [9], the ELF decoder needs to return a region mapping containing a *region* $\mapsto$ *dataspace* pair for each ELF program segment.

The current implementation of the ELF decoder is also the dataspace manager which exports the program segment dataspaces. This dataspace manager hierarchy is depicted in Figure 2.2.

Once the implementation of advanced dataspace manager features has caught up with the design, then a level of indirection can be eliminated: regions containing read-only program segments could be backed directly by the ELF file dataspace; and regions containing initialised writeable program segments could be backed by a dataspace from the anonymous memory dataspace manager which was a lazy copy of the file dataspace. Such a structure is shown in Figure 2.3. Of course the ELF decoder would still be required to set up such a region mapping, but its job would be complete before the new program is started.

Reading the ELF file header and segment headers to split the file into program segments is the most straightforward part of the implementation. The difficulties arose from the requirement to export dataspaces to multiple clients with each client possibly using a different ELF files. If the two clients use the same ELF file, then naturally it was desirable to have that ELF file open only once. The ELF files are dataspace, so they was attached to reserved regions of the address space.

There are two attributes of an ELF program segment that affected how the segment was handled. Firstly, the segment may be read-only or read-write. Actually the ELF file can specify any combination of read, write and executable flags, but L4 (and the Intel virtual memory architecture) only differentiate between read-only and read-write pages. A dataspace fault for a read-only segment is fulfilled by mapping a page directly from the ELF file (ie from the region which has the ELF file attached). A dataspace fault for a writeable segment is fulfilled by calling the copy-on-write support library.

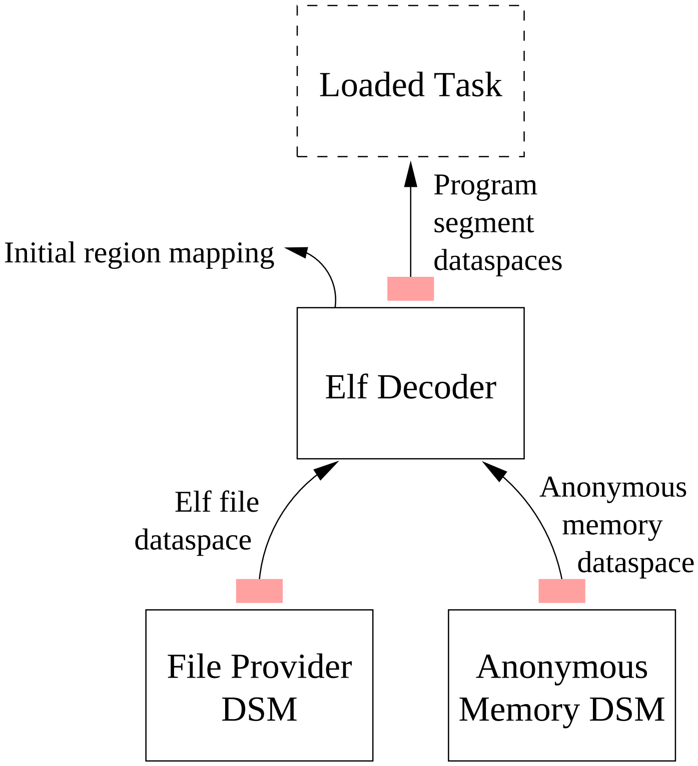

Figure 2.2: The dataspace hierarchy in my implementation of the ELF decoder.

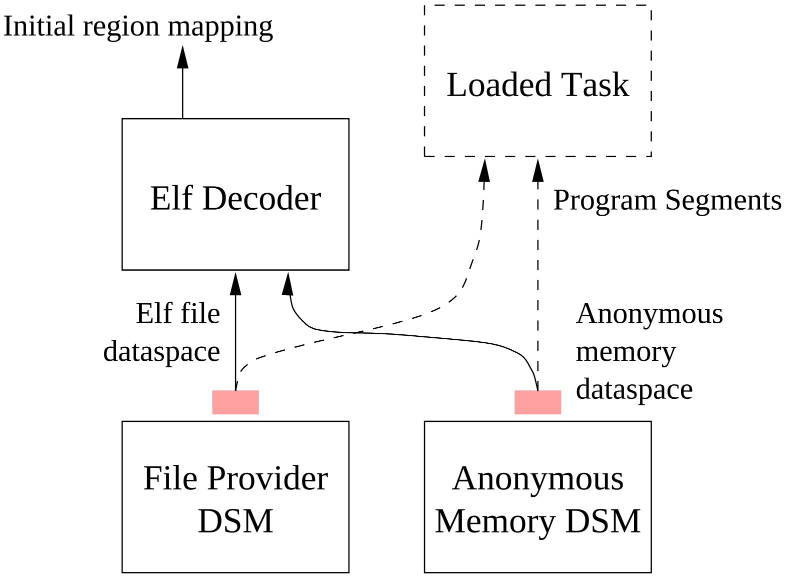

Figure 2.3: Removing a level of indirection from the dataspace manager hierarchy.

The second interesting attribute of an ELF program segment is its size. The image of a segment in the ELF file can be smaller than the segment in memory. The segment in memory must be initialised with the image from the file, then any remaining space in the segment must be initialised with zero. This is used to support traditional `.data` and `.bss` sections. These two sections are merged into a single ELF program segment. The `.data` section contains initialised data, so it is put at the start of the segment and is initialised from the segment image in the ELF file. The `.bss` section should be initialised with zero, so it is put straight after the `.data` section in the segment. The image of the segment in the ELF file is only as long as the `data` section, so the `bss` section ends up getting zero filled.

Upon receiving a dataspace fault to a writeable segment, the ELF decoder determines whether the fault address should be initialised from the file or with zero. It then calls the appropriate method in the copy-on-write support library — either `get_page_copy` which returns a copy of a page from a blueprint page (which in this case in in the ELF file), or `get_zeroed_copy` for a zero-initialised page. If there is a second fault to the same page for the same client, the copy-on-write support library will not reinitialise the page but will return the same copy that was previously mapped to the client for this address.

Notice that this implementation of the ELF decoder allows code segments to be automatically shared if the same executable file is loaded twice. It lazily initialises data pages. Indeed it lazily loads code pages from the file since it only faults on an executable file page when it needs that page to fulfill a dataspace fault.

2.3.4 The Region Map Manager

The region map manager must be the lava nucleus pager for a region mapper thread in multiple client tasks.

My implementation of the region map manager actually has the region mapper statically linked to it. It maps code region mapper code pages to clients directly from its own code segment. Data pages are mapped with the aid of the copy-on-write support library. This is depicted in Figure 2.4.

Does the region mapper really need data pages provided by its pager? On the Intel implementation of the Lava nucleus, the region mapper cannot get started without having a writeable data page mapped to it. On some other implementations it might be possible to get started without a data page by opening a writeable dataspace and sending a dataspace fault short-IPC call to some dataspace manager. Thread state could be stored entirely in registers. But on the Intel implementation there is no way to save the dataspace handle across the dataspace fault IPC because all general purpose registers are destroyed by the nucleus call and all three of the IPC short message dwords are used by the dataspace fault IPC protocol.

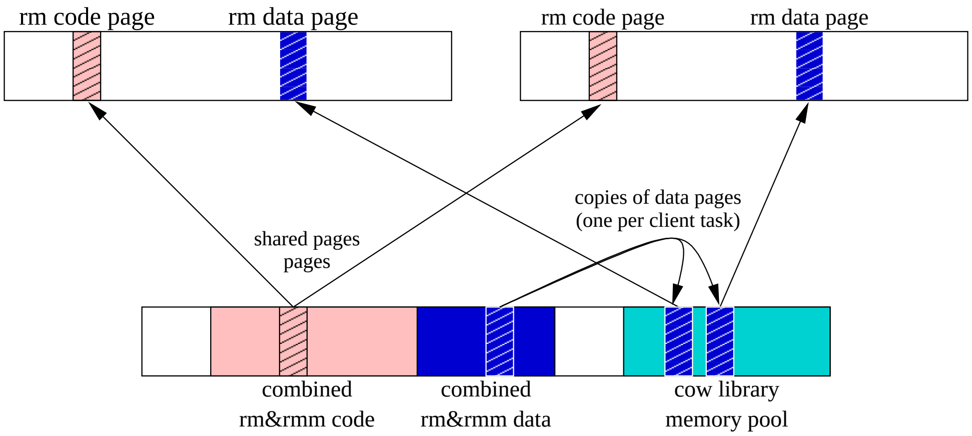

Figure 2.4: The Region Map Manager

2.3.5 The Linux File Provider

To start a new task using the Sawmill loader requires an executable file. This file must be exported by some dataspace manager.

I implemented a read-only dataspace manager which runs on L4/Linux as a Linux user process. It can export files from the Linux filesystem as dataspaces. As for the components previously described, the file provider must support multiple clients.

The file provider has a memory pool for mapping pages to clients. This memory pool is used as a cache of the files on disk. The regular Linux `read` system call is used to populate the cache. If two clients have the same file open, then it will only cache the file once.

When a cache page needs to be flushed, the file provider will first unmap the page from clients' address spaces before reading the new contents into it. If a client touches a page which was unmapped, then another dataspace fault will be sent to the fileprovider and the page can be refetched from the appropriate file.

Of course the final Sawmill system cannot bootstrap itself using executable files from this file provider, since it is running as a Linux task. Indeed the group already has a TFTP based file provider which was based on this code and on the "Grand Unified Boot Loader" (GRUB) code. However the Linux file provider is currently useful for starting Sawmill components from Linux, as described in the next section.

2.3.6 Running a Sawmill Component from Linux

I have written a sample program which uses the loader and the Linux file provider to start a Sawmill task. My sample program runs as a Linux user process. This allows a Sawmill task to be started from the Linux command prompt, for example:

```
bash$ file ./ibmip
ibmip: ELF 32-bit LSB executable, Intel 80386, version 1, statically linked, not
 stripped
bash$ runl4 ./ibmip
```

```
runl4: new task is running with task number 0x23
bash$
```

This sample program uses a library function which I wrote called `run_elf` which runs a program given an ELF file dataspace. The dataspace is obtained from the Linux file provider.

Previously the only way to start non-linux tasks was to start them at boot time. I found that using this program greatly accelerated my testing and debugging cycle because I could kill and restart non-linux tasks without having to reboot the system.

Of course this loading mechanism is also essential for a system which is not dependent upon the L4/Linux server.

Chapter 3

Debugging a Component

Developing components in Sawmill regrettably involves debugging components in Sawmill.

The Lava nucleus comes with the *Lava Nucleus Kernel Debugger* (LNKD) which has some support for debugging Lava tasks. When it is invoked, the whole system is suspended (even the system timer). The address space and registers of the active thread can be examined, and the state of any other thread can be displayed. Physical memory and nucleus page tables can also be examined, allowing the possibility of examining other tasks' address spaces, albeit painfully. Page faults and other IPC can be monitored, and there are some very limited mechanisms for restricting which IPC are monitored. One system-wide address access breakpoint can be set.

The Lava Nucleus Kernel Debugger is a good last resort when "printf debugging" fails, and it is also useful for tracking down nucleus bugs. It becomes more difficult to use in a system with many running threads when we are only interesting in a small group of threads. Using it can be a painstaking process since it is not a symbolic debugger but is very low-level.

This chapter describes how I modified the GNU debugger *gdb* [10] and the Sawmill loader to debug multithreaded Sawmill components. I used the Data Display Debugger *ddd* unmodified to provide a GUI user interface to gdb.

3.1 A New Serial Protocol for GDB

The GNU debugger supports *remote debugging* where a stub in the process being debugged communicates to the debugger through a communication channel resembling a serial link. The GDB source code refers to the kind of channel as a *serial interface*. The debugger supports several kinds of serial interfaces, including a real serial link and a TCP socket. I modified GDB to support a new kind of serial interface based on Lava IPC.

The serial interface is very simple; it must support an operation to send a character, to receive a character and optionally to send a *break* signal. The communication channel must be full-duplex, so plain synchronous IPC is not sufficient. The receive character operation must also be a blocking operation. It would be preferable not to implement the blocking by a busy wait loop. The communication channel does not need to support a very large data throughput.

I wrote an library to support what I called an *IPC pipe*, which is a point-to-point communication channel based on Lava IPC. It uses two extra threads in the task at each end of the channel: the *listener* thread and the *speaker* thread. When one of the tasks calls a function to send a character, the character will be placed in the outgoing queue. The speaker thread in that task sends an IPC for each character in the outgoing queue to the listener thread in the other task. The listener thread receives IPC and places characters in the incoming queue. When one of the tasks calls a function to send a character, the character will be obtained from the head of the incoming queue.

The above system requires bounded-buffer consumer-producer synchronisation for all accesses to the queues.

GDB was modified to link in this IPC pipe library and to support an additional serial interface which using IPC pipes. Debugging through a serial interface is initiated by the gdb command:

`target remote` *remote_address*

If *remote_address* contains the character '`:`' then it is taken as a TCP/IP address; otherwise it was taken as the filename of a serial device. Now it may also take the form '`l4-`*name*', where *name* can be looked up in the nameserver to obtain a lava thread id.

3.2 The Sawmill GDB Stub

The GDB stub is the code linked into the task being debugged which communicates with the debugger via a serial protocol.

I started with some sample code for an i386 gdb stub, but I ended up having to rewrite most of the stub when I added support for debugging multithreaded tasks.

The basic concept for the way a gdb stub operates in a single-threaded process is as follows: the user process runs unhindered until it causes an exception. The exception handler is the gdb stub, which interacts with the debugger and can report and modify process state while the process is suspended. The stub can resume the process by returning from the exception handler. Breakpoints can be set by overwriting code with trap instructions so that an exception is generated when that point in the code is reached. Note that this requires the code segment to be mapped writeable to the user task for the gdb stub to be able to use breakpoints.

For a multithreaded process, I used a similar concept for the gdb stub. I will describe four differences for the gdb stub with multithreaded support.

Firstly, when one thread generates an exception, the whole process is not stopped. It would be nice to stop the whole process, but this cannot be done without canceling or aborting any pending IPC. This should not be a problem because the user code should be sensible enough to retry an IPC which is canceled or aborted, but unfortunately most of the code that I was dealing with was not so sensible. So when one thread is stopped because of an exception, the other threads continue running.

Secondly, to avoid confusing gdb, only one thread can be in the gdb stub at one time. This means that if one thread generates an exception while another thread is in the gdb stub, then the first thread must wait until the second thread is resumed. I actually implemented this by using a dedicated thread for the gdb stub, and setting up the exception handler for each thread to do an IPC call to the gdb stub. This causes any thread which generates an exception to be blocked until the gdb stub tells it to resume by replying to the IPC call. One advantage of having a dedicated thread for the gdb stub is that gdb can communicate with the stub even while all user threads are running. This allows the programmer to press CTRL-C at the gdb prompt and proceed to set breakpoints, examine memory, or terminate the process even while all threads are still running.

The third difference for the gdb stub with multithreaded support is the way that a thread's registers are saved at the start of the exception handler. Each thread shares the same exception handler code, but this code must save all of the thread's registers in a thread-specific location (otherwise the registers from one thread would overwrite the saved registers from another thread). However the code cannot determine in general which thread is running it except by doing an `ln_myself()` nucleus call. This nucleus call destroys register values, thus defeating the purpose of finding out which thread is running the code. My solution is to first save the registers in a global area protected by a spin-lock. Then the current thread can be determined using the nucleus call, after which the register values can be copied to a thread-specific area and the spin-lock can be exited. On the Intel architecture I can implement a spin lock which does not modify any registers. Provided that all threads in this task have the same priority, using a spin lock is no concern because a very short time is spent within the lock.

The fourth difference with multithreaded support is that we need to be able to view the registers of threads other than the thread in the gdb stub. This is more difficult unless the other thread is blocked waiting to enter the gdb stub also. A general solution would be to `ln_thread_ex_regs` the thread to some code which saves all the registers for the gdb stub, then `ln_thread_ex_regs` it back to where it was before. But if the thread was blocked on an IPC then the IPC would be canceled or aborted which is a problem as described above. So this general solution will cause the user program to fail. I resorted to finding only the values of those registers which can be found passively. The instruction pointer, stack pointer and flags can be obtained passively from the `ln_thread_ex_regs` nucleus call. It is also very useful to obtain the frame pointer `EBP` so that gdb can see the stack frame of the thread. So if the thread is blocked on some IPC (as is often the case) then I can obtain the base pointer value from the top element on the stack where it is usually pushed. This is just a guess, but a better guess than any other value since it is almost always correct. So the state of other threads can be reported to gdb, including the code position and stack frame, except for the values of register variables.

The gdb stub needs to install its exception handlers for each thread. On the Intel platform, there is a privileged instruction which loads a special purpose register with a pointer to an *interrupt descriptor table* (IDT) which contains pointers to exception handlers. The Lava nucleus virtualises this instruction, allowing each thread to have its own interrupt descriptor table. So each user thread must call an initialisation function in the gdb stub when it is first started to install the exception handlers. This has been added to the appropriate functions in the thread library.

3.3 Debugging Support in the Loader

The standard way to use a gdb stub is to statically link the stub with the program to be debugged. This means that regular binaries cannot be debugged; a program needs to be relinked in order to debug it.

In the loading scheme for Sawmill descibed in the previous chapter, a region mapper is placed in the address space of each loaded task. I linked the gdb stub with the region mapper so that it automatically exists in each task which uses this region mapper. This allows any program to be debugged, simply by starting it with the right region mapper.

It might have been even nicer to load the gdb stub into the new task as an extra program segment rather than paging it in with the region mapper. This can be done by putting the gdb stub into its own executable file. Then when the executable file for the program to be debugged is decoded, the gdb stub's executable can also be decoded and the two resulting region mappings can be merged. However the region mapper needs to be aware of the gdb stub for reasons descibed below, so I found it easier just to link the gdb stub with the region mapper.

There is one class of exceptions which cannot be caught by the gdb stub running under the Lava nucleus: page fault exceptions. Page faults are handled via a page fault IPC being sent to a thread's pager, which for a user thread in a Sawmill task will be the region mapper. The region mapper tries to convert the page fault to a dataspace fault, but this might fail. The dataspace manager might not fulfill the dataspace fault for some reason. Indeed the address of the page fault might not correspond to any region with a dataspace attached. So when the region mapper encounters a bad memory fault, it needs to inform the gdb stub.

The region mapper will actually cause the faulting thread to simulate a page fault exception. The region mapper `ln_thread_ex_regs` the faulting thread to some code which pushes the original instruction pointer on the stack and jumps to the gdb stub's page fault exception handler. When this thread returns from the exception handler it will retry the same instruction that caused the memory fault, as required.

The `runl4` command was modified to accept an extra flag: '`-g`'. This flag indicates that the program should be run for debugging rather than being run normally. The region mapper in the new task will stop and wait for GDB before jumping to the program entry point. Once GDB has opened a communication channel with it, the first user thread of the task is "in the debugger" and will not proceed until instructed to do so by the user. This allows the user to set breakpoints before the program begins running.

3.4 A Brief User Manual for DDD on Sawmill

This chapter attempts to provide a brief overview of using the Data Display Debugger (DDD) to debug Sawmill tasks. Familiarity with regular use of DDD or at least with GDB is assumed

To begin a task for debugging, use the `runl4` utility with the `-g` flag and start the debugger:

```
bash$ runl4 -g ./ibmip
runl4: new task is running with task number 0x23
runl4: now run our gdb and give it the command ‘target remote l4-23’
bash$ ddd ./ibmip &
```

When the DDD windows pop up, copy the ‘target remote’ command from the output of `runl4` and paste it at the gdb prompt:

```
(gdb) target remote l4-23
```

Now the debugger is ready to go, and the task being debugged is stopped in the region mapper's initialisation code just before calling the program's entry point. You probably want to start with setting a breakpoint and continuing the program.

I will draw attention to the following features of the debugger. The applicable gdb console command is listed on the left, but most commands have an equivalent button, menu option or popup window.

`kill`	Terminates the task being debugged, closing the dataspaces which are currently attached to regions in its address space. The task may be re-run using `runl4 -g` and then the `target remote` command can be issued again at the gdb prompt.
`detach`	Detaches the debugger from the task being debugged.
`quit`	Terminates the task being debugged as for `kill`, then exits the debugger.
`info threads`	Lists the status of all active user threads. In DDD there is also a window that can be kept open to show this information. The function and source line of each thread is listed, along with the parameters to that function.
`bt`	Shows a back-trace of the function call stack for the currently chosen thread. The call stack of different threads can be viewed using the `thread` command.
`thread` *n*	Chooses thread *n*. This does not allow you to switch which thread is being single-stepped because only the thread which is currently stopped in the debugger can be stepped or continued. To singlel step a different thread you must first stop it with a breakpoint. Clicking on a thread in the thread status window has the same effect as using this command.

`step/stepi`	single-step one source line or one machine instruction respectively of the thread currently stopped in the debugger.
`next/nexti`	same as step and stepi, but function calls are stepped over rather than being entered.
`finish`	cause the currently stopped thread to execute until it returns from the current function.
`break` *address/function/ line number*	sets a breakpoint that will stop *any* thread at the specified address or source code position. It is much easier to click at the left of the source code line and press the 'break' button.
`thread` *n* `break` *address / ...*	sets a breakpoint only for the given thread at the specified address or source code position.
`cond` *n condition*	sets a condition for the specified breakpoint. There is also a dialogue box for reviewing and editing all breakpoints and conditions.
`graph display` *variable*	displays any program variable, showing the structure of `structs` and allowing pointers to be followed using the mouse.

Also a disassembly of the program can be displayed as well as the source code by choosing Source / Display Machine Code from the menu.

Chapter 4

Managing Address Spaces of Linux User Tasks

This chapter describes a strategy for migrating the management of user address spaces out of the Linux kernel.

In L4/Linux, as for monolithic Linux, the Linux kernel manages the address space of each user process. There is a page table for each user process maintained by the kernel.

The virtual memory framework for Sawmill has already been described. This can be used in the management of the address spaces of Linux user processes. The process could be loaded using the Sawmill loader, which provides for code and data to be backed by appropriate dataspaces. Stack and heap can be supported by dataspaces from the anonymous memory server.

4.1 Accessing User Memory for Handling System Calls

4.1.1 Autonomy of User Tasks

Many Linux system calls require that the kernel accesses user memory. In monolithic Linux, the user address space is mapped into the kernel's address space, so the kernel can directly access user memory. However in L4/Linux, the kernel does not map the user address space into its own address space. Instead, when it wants to access user memory, it looks up in its pagetables to find the corresponding physical frame and it accesses the physical frame directly.

Both of these methods are undesirable in Sawmill, where there is no centralised system call handler. Many different Sawmill components might service each user process and we probably do not want to trust each Sawmill component to have unlimited access to user processes' address spaces.

It would be nice if a user task could choose which components to trust and how far to trust them. To do this, the task needs some kind of autonomous control over its own address space. This can be achieved using the virtual memory framework in Sawmill, where a task's address space is managed by a region mapper and supported by various dataspace managers, as previously described.

So the virtual memory system of Linux needs to be migrated to a system where each task is responsible for maintaining its own address space using its region mapper. Naturally, to continue support for the Linux binary interface, this extra management will be performed by the emulation library and the region mapper.

4.1.2 Controlling External Data Accesses

Remember that system calls are handled by an emulation library in the task which converts the system call into an IPC to an appropriate service. The current emulation library sends all system call IPCs to

A dilemma is encountered when designing how external services can access the appropriate data for fulfilling system calls without trusting them more than necessary.

It would be easiest for the emulation library to control external data access by explicitly sending and receiving the appropriate data in the IPC call to the external service. However the existing code in the Linux kernel is centred around kernel-initiated memory access, and it would take quite a lot of work to rewrite all this code.

Several specific system calls cause particular trouble. For the `ioctl` system call, it might be very difficult for the emulation library to determine which addresses contain parameters to the system call, due to the variety of device-specific `ioctl` interfaces. Special care needs to be taken with `readv` and `writev` to avoid unnecessary copying.

Two methods for handling external data accesses were discussed in the group: sending the appropriate data with the IPC from the emulation library, or alternatively having a dedicated thread in the user task to fulfill external data access requests. The former approach suffers from the disadvantage mentioned above, particularly with the use of existing Linux kernel code centred around kernel-initiated user memory access. With the latter approach it is difficult for the emulation library to restrict external data access appropriately.

I propose a hybrid approach: rather than dedicating a thread to fulfilling external data access requests, the thread which traps into the emulation library and makes the IPC can handle external data access requests too, as shown by the following pseudo-code.

```
prepare_syscall_mesg(&sendmesg);
while(true) {
    IPC_call(service, sendmesg, &rcvmesg);
    if (SYSCALL_COMPLETE(rcvmesg))
        break;
    handle_extern_data_access(rcvmesg, &sendmesg);
}
```

This method allows existing code based on kernel-initiated user memory access to work without major modifications, while still giving the emulation library the chance to control external data access requests in the context of each system call.

However it is still not possible to handle `ioctl` safely in general with this method. For example, suppose a new linux driver is added to the system after the emulation library has been implemented, and suppose this new driver supports an `ioctl` whose third argument is a struct containing a pointer (the third argument to the `ioctl` system call has no fixed structure; it depends on the particular Linux drivers associated with the specified file descriptor). Now when this `ioctl` is used, the driver will request to access the user memory referenced by this pointer. This should be a valid access because the user code has explicitly told the driver to access this memory by calling this `ioctl`. However the emulation library has no way of knowing whether this is a valid access because it doesn't know the format of this argument to `ioctl`.

I argue that it is acceptable to not support strange `ioctl`s which are introduced into Linux after the emulation library has been developed. Indeed when new Linux code is ported into a Sawmill component, the emulation library can be updated to support any new `ioctl`s.

I have made an initial implementation of this change to the system call handling in the emulation library. My implementation does not perform any checks on user memory accesses; to do so requires the system call to be decoded in the emulation library whereas it is currently decoded in the Linux server. So now the emulation library supports an external user memory access protocol, while still remaining compatibly with the existing Linux server.

4.2 Migration Strategy

4.2.1 What Needs to be Changed

Putting a region mapper into Linux user tasks is one step towards removing user address space management out of Linux. However this step alone will break L4/Linux. Here is a list of what needs to be changed:

- A region mapper needs to be put into Linux user tasks to allow them to use the Sawmill virtual memory framework.
- A mechanism for external accesses to user memory needs to be established, as described in the previous section.
- The `fork` and `clone` system calls involve duplicating the task's address space. Since the task itself is the only entity which knows about its address space, it needs to provide the initial region mapping for the new task.
- The `exec`, `mmap` and `munmap` system calls involve modifying the task's address space. These need to be converted into the appropriate dataspace operations by the emulation library.
- The user address space can be moved away from being paged entirely by the Linux kernel to being paged by various dataspace managers.

4.2.2 Small Steps

From a software engineering perspective it would be undesirable to perform all of these changes at once, because they form quite a huge modification to the existing system when combined. Instead, it is desirable to take smaller steps with the property that the system can be run, tested and debugged after each step.

These are the steps that I proposed to separate user address space handling from the Linux server, as refined by discussions with other group members.

1. Change the pager of user address spaces in the Linux kernel to export a dataspace and accept dataspace faults as well as page faults. A user process's address space is considered to be a single huge dataspace which should be attached to a region covering the whole address space.

2. Modify the way a Linux user task is loaded so that it has a region mapper backed by the region map manager, and uses an initial region mapping which contains the single dataspace described above.

3. Add the mechanism for external accesses to user memory to the emulation library. The Linux server contains several functions which are used to access user memory. These functions need to be changed to use the new user memory accessing protocol.

4. Allow the Linux kernel pager to export files as dataspaces to be used by `mmap`. This is a small extension since an `mmap`ed file will be a subrange of the huge single dataspace mentioned above.

5. Add handling of `mmap` and `munmap` into the emulation library. These operations should result in dataspace open/attach and detach/close operations respectively. Also the huge single dataspace from the Linux kernel will need to be reattached to avoid clashes with `mmap`ed files.

6. Put `fork/clone` handling into the emulation library. This involves the emulation library co-operating with the Linux server (in the future with the Linux process manager) to create a new Linux user task, with an initial region mapping provided by the emulation library. The initial region mapping should match the original task's current region mapping, with each dataspace appropriately copied or shared. Other process state may need to be copied to the new task as well, such as open file descriptors. To begin with, this can be handled as before by the Linux server.

7. Change `exec` to use the Sawmill loader described in Chapter 3. The emulation library needs to close and detach all dataspaces, get the executable file decoded, and use the initial region mapping provided by the executable decoder to set up the address space again. This final step cannot be performed until there is some reliable file system which is not dependent on the Linux server, or until the Linux server can export arbitrary files as dataspaces, so that the ELF executable can be obtained. The dataspace managers involved must support the dataspace **copy** and **share** operations which are used by `fork`.

After these steps, Linux user tasks will be running without having their address spaces managed by the Linux server.

Chapter 5

Conclusions

During this year there has been a lot of progress made with Sawmill, but there is still a lot more development required. The Sawmill loader is now working and in use. This makes it possible to load Sawmill tasks interactively, which greatly accelerates the test/debug/recompile cycle in the further development of Sawmill components.

Furthermore the GNU debugger is working nicely on Sawmill and it has proven quite useful to me in debugging an experimental port of a network device driver and TCP/IP stack. I hope that it will also help to speed the development of Sawmill services.

A course has been plotted for removing the virtual memory management from the Linux server, which will be a big step in the push to move all of its functionality into Sawmill services.

Bibliography

[1] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers and Susan Eggers, "**Extensibility, Safety and Performance in the SPIN Operating System**", *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15), pp. 267–284*, University of Washington.

[2] Margo I. Seltzer, Yasuhiro Endo, Christopher Small and Keith A. Smith, "**Dealing With Disaster: Surviving Misbehaved Kernel Extensions**", *Proceedings of the 1996 Symposium on Operating System Design and Implementation (OSDI II)*, Harvard University.

[3] J. Mark Stevenson and Daniel P. Julin, "**Mach-US: UNIX On Generic OS Object Servers**", *Usenix Technical Conference*, CMU, 1995.

[4] L. Torvalds et al., "**Linux Kernel Source, version 2.2.13**" *available from* `ftp://ftp.kernel.org/pub/linux/kernel/v2.2/linux-2.2.13.tar.gz`, 1999.

[5] Mohit Aron, Jochen Liedtke, Yoonho Park, Kevin Elphinstone, "**A framework for VM diversity**", *Technical Report*, IBM Research, 1999.

[6] Various authors, "**FreeBSD**" *see* `http://www.freebsd.org`.

[7] Sun Microsystems, "**ChorusOS Real-Time Operating System White Paper**", *available from* `http://www.sun.com/software/chorusos/wp-emb.telecom.platform/index.htm`,

[8] Yousef A. Khalidi and Michael N. Nelson, "**The Spring Virtual Memory System**", *Technical Report TR-93-9, available from* `http://www.sun.com/research/techrep/1993/abstract-9.html` , 1993

[9] Sun Microsystems, "**System V Application Binary Interface, version 3.1**".

[10] John Gilmore, Stan Shebs, "**GDB Internals**", revision 1.1, Free Software Foundation, 1999.

[11] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young, "**Mach Kernel Interface Manual**", *Unpublished manuscript from the School of Computer Science, Carnegie Mellon University,* 1990.