Course of Study: Software Engineering

Examiner: Prof. Dr. rer. nat. K. Lagally

Supervisor: Dr. rer. nat. T. Schöbel-Theuer

Thesis No. 2505

Stateless CPU And Memory Multiplexing In Athomux

Marcel Kilgus <marcel@kilgus.net>

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

Commenced: 2th June 2006

Completed: 2th December 2006

CR-Classification: D.4.0, D.4.7

Abstract

Athomux, which is a prototype of a new operating system, previously only ran under a different host system, namely Linux. The work described herein made Athomux a self-hosting system, not depending on any outside software.

This involved creating a scheduler and a virtual memory system. The new components are supposed to be stateless. In this context (pseudo-)stateless means that the components can be instantiated and destroyed without any loss in functionality or data.

Table of Contents

Index of Texts

List of Figures

1 Introduction

Thomas Schöbel–Theuer has proposed a new design and programming method for operating system kernels[1] called Athomux. It is described as a LEGO-like approach to creating operating systems, with the goal of achieving high modularity, reusability and flexibility of components. Similar to a LEGO toy brick system the basic bricks of Athomux are supposed to be easily combinable in any sensible way.

There are only 2 basic concepts called “bricks” and “nests”. Staying within the LEGO analogy, bricks are bricks and nests are the protruding interlocking studs used to hold the pieces together.

So, essentially bricks are pieces of code and nests are a form of virtual address space used to represent all sorts of different data. Bricks can have both input and output nests. The output nests provide the operations on the data that the input nests can call. Several input nests can be connected to an output nest in any way that makes sense.

1.1 Motivation and Objective

The goal of this work is to take the existing Athomux implementation, which runs under the host operating system Linux as user processes and create the necessary bricks to make the system self hosting. This involves, in addition to all the code involved in the boot process itself and some minimal hardware drivers, at least a CPU brick, which manages the tasks and preemptive multitasking, and an MMU brick, which manages the virtual memory address spaces of the tasks.

Once these work, the existing POSIX sub-system[2] should be made to work on top of the new system, ideally providing the standard Linux system call interface to applications.

1.2 Structure of this Work

The thesis is organized as follows:

- Chapter 2 shows the overall architecture and high level interaction of the existing and newly created bricks.
- Chapter 3 describes the implementation details and pitfalls encountered during the implementation.
- Chapter 4 provides a quick start guide about the actual usage of the new system and describes the tools used for the development.
- Chapter 5 gives a rough overview of the performance of the Athomux kernel.
- Chapter 6 finally concludes the work with a summary and suggestions for further development.

2 System Architecture Overview

Following is a rough schematics that shows the different bricks used in the native Athomux implementation and how they are connected to each other. The diagram omits most bricks not directly involved in the subject of the presented work. The structure is basically derived from the developments of Florian Niebling[2] and the reader is referred to his thesis for those parts that are not covered by this work.

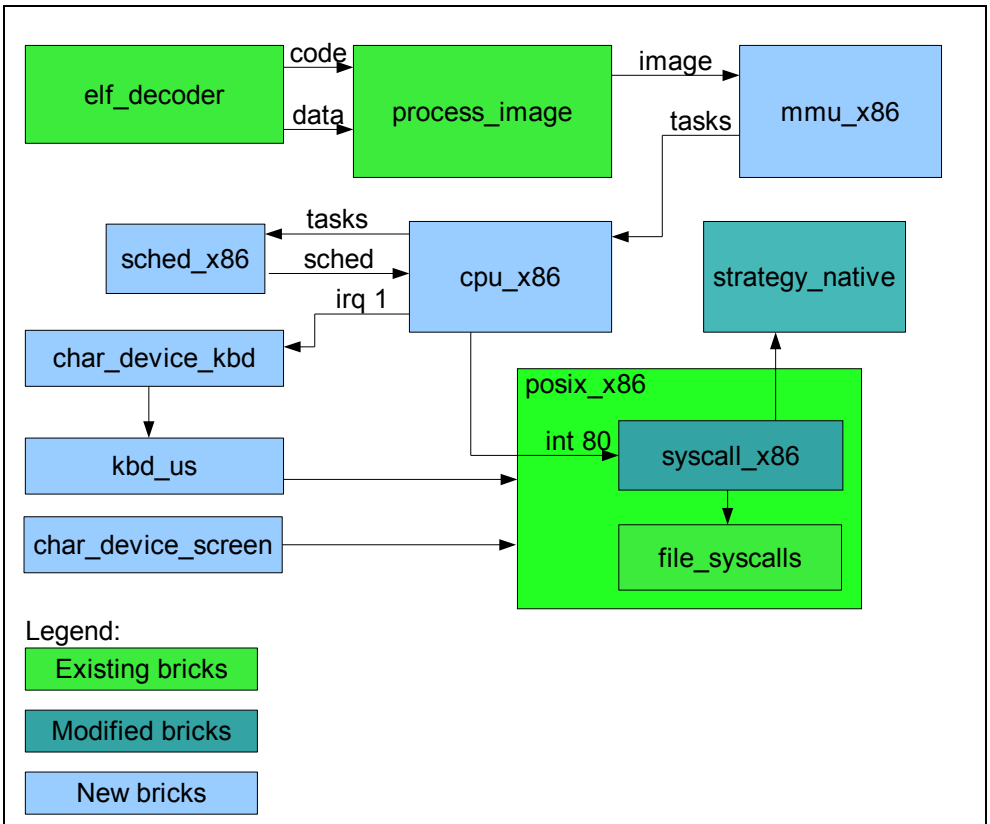

Figure 1: Architecture overview

Short introduction: the central brick is the *cpu_x86* brick, which manages all the user tasks and, with help of the *sched_x86* brick, does the preemptive multitasking. The *strategy_native* brick is responsible for building and connecting all the other bricks during system startup, task creation and destruction. The *elf_decoder* unravels an executable file in ELF file format, which the *process_image* brick puts together again to form the process's virtual address space. The *mmu_x86* brick finally mirrors this space in actual physical memory

using the MMU hardware. The *posix_x86* brick provides the Linux system call interface to the user process. The *char_** bricks are simple hardware drivers.

2.1 cpu_x86

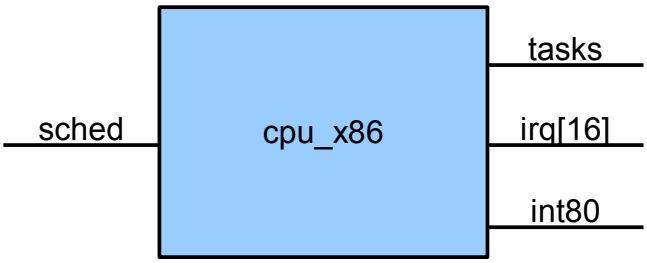

Figure 2: cpu_x86 brick schematics

This is the main brick regarding Athomux tasks and multitasking. It is the interface to most of the actual CPU hardware, including interrupts and exceptions. During startup it initializes the task list and all interrupt vectors. The hardware IRQs 0-15 are routed to the respective outputs of the brick (unlike all other connections execution flow of exceptions travel from output to input as this way several consumer inputs can be connected to one IRQ output).

The software interrupt 0x80 is also routed to its own output, as this is the standard Linux interface for system calls.

The CPU brick itself connects to the timer interrupt which is then used for preemptive multitasking. When a task switch is supposed to occur, the CPU brick sends a request to the *sched_x86* brick, which in turn asks the CPU brick for all task related data to make its decision on which task to activate next. This way the CPU brick can stay the same and the task scheduling strategy can even be "hot swapped" with a different algorithm.

The brick also receives the general protection/paging fault CPU exception and passes it on to the *mmu_x86* brick. This is however not done using the standard connection scheme, as the instance to be activated needs to be decided in the context of the currently active task.

All internal data is held within the "tasks" nest, which has 2 sections, one with the actual task data and one with the meta data like the currently active task and so on.

2.2 sched_x86

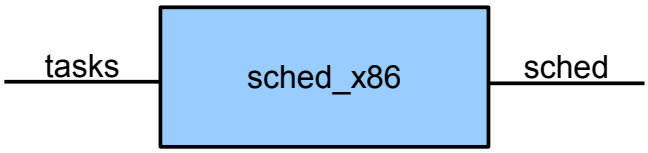

Figure 3: sched_x86 brick schematics

This is the scheduling strategy brick that is called by *cpu_x86* when a task switch is supposed to happen. It is responsible for selecting the next task to switch to. The current code implements a very simple round robin system where every runnable task gets equal time without any priorities or other criteria whatsoever.

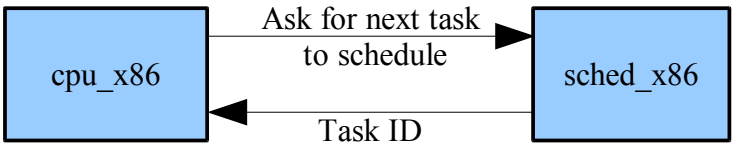

Figure 4: Brick interaction during task selection

2.3 mmu_x86

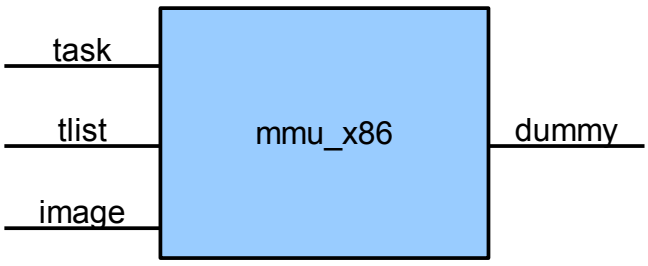

Figure 5: mmu_x86 brick schematics

This brick is the interface to the CPU's virtual memory management. As every process has its own virtual address space and thus its own *mmu_x86* instance, the brick is also trusted with initiating task creation and destruction. On instantiation of a new *mmu_x86* brick the *cpu_x86* brick is commanded to create a new task in its internal structures. Upon destruction the task is deleted accordingly. Changes to the physical memory are currently not written back to

the image nest, mainly for performance reasons. An operation that triggers the write back could however be implemented fairly trivially if the need every arises.

The brick handles both the *execve* (an entirely new task is created) and *fork* (the current task is duplicated) case.

In case of *execve*, a copy of the clean kernel MMU page tree is made, which does not yet include any pages in the user part of the virtual address space. Then a minimum user stack environment is created within the user space and the arguments to the main procedure of the program to be run (argc, argv, envp) are pushed onto this stack with standard C calling convention. The stack pointer is initialized to the newly created stack and the instruction pointer is set to the task entry point that has previously been extracted out of the ELF binary.

In case of a *fork*, a copy of the MMU page tree of the current process is made, marking both page trees read-only and with the software-defined "copy on write" flag during the copy process. The usage counter of all physical pages encountered during the copy is also incremented. Then the new task structure is initialized with saved copies of the stack pointer, base pointer and instruction pointer of the current task.

In both cases a private kernel level stack is created in the end, then the new task is marked as runnable and execution is handed back to the calling brick.

While the processes are running, the *mmu_x86* brick is also responsible for handling the page fault exceptions handed over from the *cpu_x86* brick.

When a page fault is generated for a page that is not present in memory, then a new page of physical memory is allocated and linked into the virtual address space at the address where the fault occurred. This is then filled by calling the brick connected to the image input, which would normally be a *process_image* brick. After that, the brick hands execution back to the offending process, which can now continue to work as if nothing had happened.

If the exception was not a page missing fault but a general protection violation, the entry in the MMU tree is checked for the "copy on write" flag. If this is not present the whole OS or preferably just the offending process is terminated.

If on the other hand the page was marked "copy on write", then the usage count of the page is checked. Is this set to one, then the page is just marked as writable again and execution continues as normal. Otherwise a new physical memory page is allocated and filled with a copy of the page that was marked

"copy on write". This is linked into the page tree of the current task, unlinking the original page in the process and decrementing its usage counter.

2.4 strategy_native

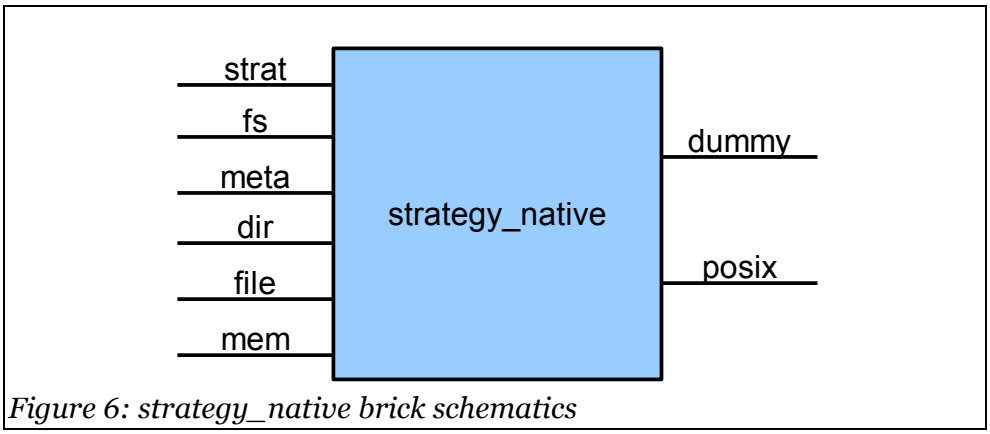

Figure 6: strategy_native brick schematics

In the current design of the upper process levels, which is mostly derived from the Linux user land implementation of Athomux [2], the *strategy_native* brick fulfills two functions: on one hand it is responsible for the system startup of most bricks, connecting everything necessary to form a runnable process. These connections and relations are not shown in the diagram. On the other hand, it also directly handles some syscalls that get passed from the *syscall_x86* brick, most of which have to do with task management (fork, execve, wait4 etc.), but some file system calls are handled here, too (open, close, pipe, dup etc.).

2.5 posix_x86

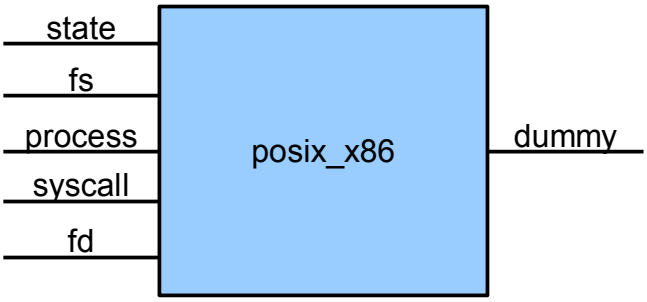

Figure 7: posix_x86 brick schematics

This is basically just a wrapper brick around the *syscall_x86* and *file_syscalls* bricks.

2.6 syscall_x86

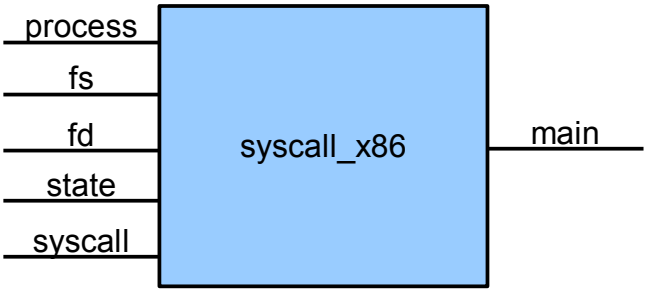

Figure 8: syscall_x86 brick schematics

This block is directly connected to the int80 output of the cpu_x86 brick. Interrupt number 0x80 is traditionally used on Linux systems for calls from user processes into the operation system kernel. Arguments are passed in registers and optionally memory.

The brick itself does only handle very few system calls itself, mostly it passes them on to other more specialized bricks, like strategy_native (process and file management) and file_syscalls (file I/O).

2.7 char_device_kbd

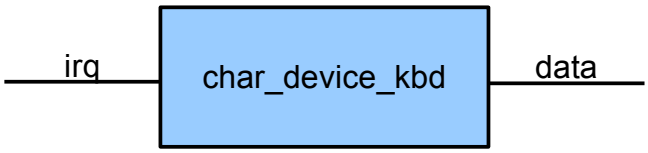

Figure 9: char_device_kbd brick schematics

A very simple keyboard device driver. The brick processes the keyboard interrupt sent from the CPU brick and puts the read scan codes into an internal queue. The contents of this queue is made available through the data output.

2.8 kbd_us

Figure 10: kbd_us brick schematics

This brick translates the keyboard scan codes it gets from the kbd input into ASCII characters according to the US keymap. The result is made available at the fd output. Currently the translation is very basic and only includes the most important keys.

2.9 char_device_screen

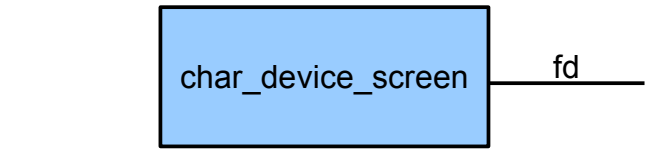

Figure 11: char_device_screen brick schematics

This is a driver for the standard 80x25 VGA text screen display. The stream of characters sent to the output fd is shown on the screen, scrolling it if necessary.

No real terminal emulation is done at this point and should probably be done in a separate brick anyway.

2.10 device_ide

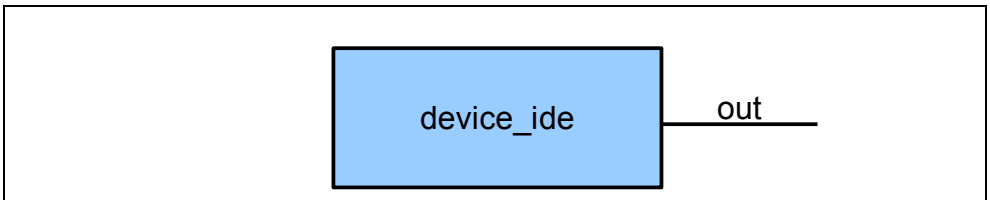

Figure 12: device_ide brick schematics

This brick provides minimal IDE support as detailed in the ANSI X.3321 standard. It uses simple PIO (programmed I/O) for operation instead of the more complex DMA (direct memory access) mode. The contents of the primary disc on the first IDE channel is provided on the “out” nest.

The brick is working but not employed in current Athomux builds.

3 Implementation

3.1 Boot Process

Several solutions for the basic boot process were considered, but in the end it was decided to go with the “Multiboot Specification” as implemented by the GNU GRUB boot loader. GRUB is readily available, free, and can handle many different storage devices and file systems.

Development was planned to be done on virtual image disc files that can be mounted by Linux and used by the different PC emulators. Originally it was intended to use hard discs images for this, but that was ultimately scrapped for several reasons:

- It doesn't seem to be possible to install GRUB directly on a mounted image of a hard disc, as it tries to detect the BIOS drive number for the disc which of course is not possible in this situation. To install GRUB, one would have to use a PC emulator, boot some Linux version with it and install GRUB within the virtual box.
- Hard discs need to be partitioned, which further complicates things.
- Most PC emulators (QEMU, Bochs, VirtualPC, VMware) use hard disc image formats that are often incompatible with each other, making quick testing with several different emulators rather difficult.
- It is rather cumbersome to transfer a created image to a real PC and make it work.

So instead it was decided to use floppy disc images as the main boot solution:

- Linux can easily mount a floppy image, making replacement of the kernel very easy.
- All PC emulators can mount a standard floppy image.
- A floppy image can easily be transferred to a real floppy disc and booted in any PC that is still outfitted with an ordinary floppy drive.
- A floppy image can also be used to create a bootable CD ROM for PCs that don't have a floppy drive.

- After installing GRUB there is still over 1,2MB of free space, which is currently enough to fit the whole Athomux kernel (stripped from debugging information) into it.

Should the floppy disc ever run out of space the proposed solution is to switch to CD-ROM ISO images, which are about equally easy to handle but provide vastly more space.

After GRUB has loaded the kernel to the physical address specified in the ELF binary (currently 0x00100000) it hands over control of the machine to the entry point multiboot_entry in x86_boot.S.

3.2 Memory Management during Boot

The whole Athomux code is linked to live at address 0xC0100000, but is loaded to 0x00100000 by GRUB. Therefore the code at this point must not make any absolute jumps or use the stack or anything. The very first thing it does do is to create a minimal MMU page tree which mirrors the first 4MB of physical memory at virtual address 0xC0000000 (defined as KERNEL_VIRT_BASE), enable paging in the CPU and then jump to itself in this upper "kernel-space" virtual memory region.

As no memory allocations can be done yet, the MMU page tree is constructed at a fixed position in low memory, namely at BOOT_PAGE_DIR (arbitrarily defined as 0x1000). The lowest megabyte of memory is, due to the history of the PC, pretty fragmented and currently otherwise not used by the Athomux kernel.

3.3 Memory Model

Earlier revisions of the operating system used a segmented memory model to distinguish between kernel and user memory space. Both the kernel data and code segments were based at address 0xC0000000, while the user space data and code segments had a base of 0 with a size of 0xBFFFFFFF. This model was directly adopted from Linux 1.0. It has a few advantages, for example the intermediate MMU step described above in section 3.2 is not necessary as physical addresses equal virtual addresses withing the kernel segments, but also

some disadvantages, one of which is that the GDB debugger cannot handle segmented memory models and a special QEMU build had to be created[3] to get a somewhat functional debugging environment. Also accesses to user code and data from within the kernel were a bit harder as they had to be done using wrapper functions.

Current builds now use a slightly modified flat memory model. As before the virtual address space is divided into user and kernel space. The user space occupies the first 3GB of virtual memory, the kernel space the last GB (address 0xC0000000 and above). But unlike the first version all 4 defined segments (kernel code and space plus user code and space) are defined with a base address of 0. The only difference between them is that kernel segments run in protection ring 0 instead of 2 and the user segments are limited to 3GB instead of 4GB. The latter is not even necessary if memory protection is done right on MMU page level but on the other hand it can't do any harm either.

All of the physical memory is mapped into kernel space, which means that currently a maximum of 1GB of physical memory is supported. This is consistent with current Linux implementations, the only difference being that Linux can still address more than 1GB of memory by special mapping tricks[4].

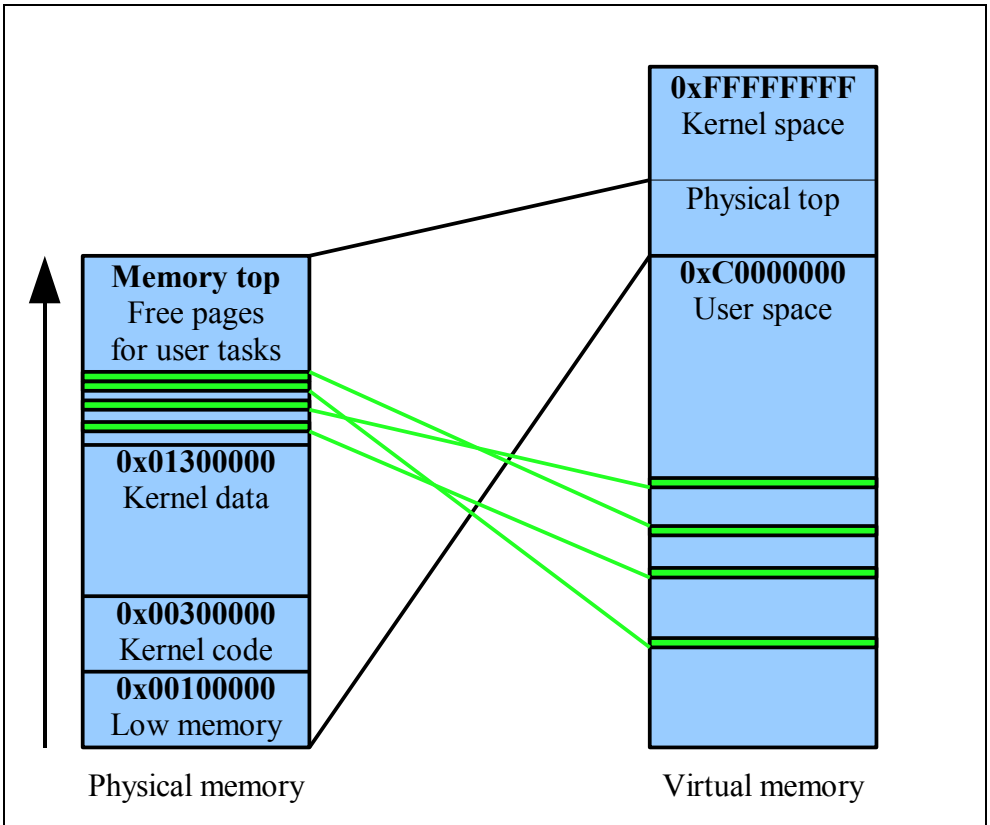

Figure 13: Athomux memory map

Upon entry into the kernel the segment registers are loaded with the kernel segment selectors, thus all code inside the kernel has access to everything. The MMU page tree of the current process stays active, this way the kernel can easily access all user data at the same virtual addresses the user task sees them (addresses 0 to 0xC0000000) but also all physical memory (addresses 0xC0000000 to 0xFFFFFFFF). The latter is necessary for allocating new memory pages and CPU related data structures, for which the physical addresses must be known.

Macros exist to aid in the virtual-to-physical and physical-to-virtual address mapping. A given physical address can be accessed by using the VIRT_MEM(address) macro, which returns a virtual address the kernel can use. The other direction is defined by the PHYS_MEM(address) macro.

3.4 Kernel Memory Allocations

All current Athomux implementations make at some point use of the malloc() and free() functions for serving their memory needs, even though this is against the Athomux philosophy. But the primary goal while developing the first steps of the native Athomux code was to leave existing code working as well as possible, only small steps and iterations could be allowed to ensure that everything is still working as expected while development goes on. Therefore compatible memory management functions for the existing code had to be provided.

Existing malloc() implementations of the different libc libraries were often too entangled within their libraries or environment to be of good use. Plans for writing new simple and list based malloc() routines were evaluated but eventually scrapped when Doug Lea's generic malloc implementation was found on the web [5]. This is a single C file, literally littered with preprocessor macros but incredibly versatile and configurable for all different environments and uses.

For Athomux an early and more simple release of the malloc code was used at first (2.5.1), but switched later to the current release 2.8.3, as this has some more features like providing specifically aligned blocks of memory.

The existing code could almost be used completely unaltered. This was achieved by providing a small and simple sbrk() implementation that grants the malloc function more kernel memory until the maximum limit defined in the

x86_kernel.h file is met. All allocations are strictly served from the space designated as “kernel data” in Figure 13.

3.5 User Memory Allocations

Space for user tasks is allocated out of the remaining parts of physical memory. At boot time all remaining pages are linked together in a single linked list with an anchor in the variable free_page_list. As these pages are still unused the links are simply directly written at address 0 of each page. When a page fault occurs or the kernel needs physical pages (for building or extending an MMU page tree for example) these allocations are served out of the free page list. As the mechanism simply consists of taking the first free page and updating the free_page_list anchor afterwards, it is very fast.

Every page in this list also has a separate usage counter, residing in array mem_map allocated during memory initialization. When the usage counter reaches 0 on a free_page() call the particular page is returned to the free page list.

3.6 Tasks

Athomux tasks and POSIX tasks are actually two different structures. And while the POSIX tasks live on top of the Athomux tasks, they are not necessary or used by the actual scheduler code. This distinction on one hand is conceptually nice as it abstracts away the POSIX structure and could in theory also provide for a different process model, on the other hand it could get in the way when trying to implementing more POSIX features like signals. More research whether the structures should be kept separate or whether they should be merged is probably needed.

3.7 Task Switching

The Athomux kernel employs the hardware task switching offered by all x86 compatible CPUs. When a re-schedule is triggered, either by a system call or the

timer interrupt, the next task to switch to is determined and then a JMP instruction jumps to the task's TSS (task state segment) selector. This causes the CPU to save all registers in the currently active TSS and load the new registers from the new TSS. The MMU page directory is also switched at this point.

What is not saved however is the state of the floating point unit (FPU). This has to be done manually by the OS and Athomux does it just before the actual task switch. Unnecessarily Athomux does currently do this on every task switch, even if the task has not used the FPU. One optimization for future versions would be to employ the TS (task switch) processor flag. This flag, when set, triggers an exception (number 7 "Device not available") when a task tries to use the FPU. The exception handler is then responsible for executing the FPU context switch.

It is also notable that Athomux kernel code itself must not use the FPU without much precaution, in essence saving and restoring the FPU context before respectively after its use.

3.8 System Call Interface

Current Linux implementations feature two system call interfaces: the sysenter/sysexit functions introduced with the Pentium Pro and the traditional int 0x80 interface. The former was introduced by Intel because it turned out that software interrupts are remarkably slow on modern processors (one interrupt can actually be slower on a 2GHz Pentium 4 than on a 850Mhz Pentium III). Current glibc versions are agnostic in regards to the interface to use, as Linux maps a dynamic library call linux-gate.so.1 into the virtual address space which provides functions that do the actual work of interfacing with the kernel.

As Athomux does currently not allow dynamic linking of binaries the best option is to provide the traditional int 0x80 interface, which is well established, and use a C library that does not use the linux-gate.so.1 mechanism and that is not too complex. Both the dietlibc[6] and uClibc[7] libraries have been successfully tested so far, without any modifications.

All system calls have one call number and 5 or at most 6 call parameters. The system call number is provided in processor register EAX, the first parameter in EBX, the next in ECX, EDX, ESI and EDI respectively. From kernel 2.4 on some

system calls also have a sixth parameter, which is given in EBP. This however is currently not needed in Athomux.

Only EAX is used as a return value. If more data is needed then this has to be done over structures in memory. If the carry flag is set on return EAX represents an error code, otherwise it's the result data.

The actual binary interface of the different calls is not very well documented, except in [8] and the Linux sources themselves.

3.9 GCC

The current Athomux sources are not compatible with GCC version 4 and above. This is not specific to the native Athomux implementation, the problems are in the existing core Athomux libraries. Therefore GCC 3.3.6 has been used during the development of this thesis and will be needed to reproduce the results.

3.10 GCC Libraries

A native Athomux environment cannot use the default libraries shipped with GCC as these are supposed to work on top of an operating system. Therefore even the most basic functions like memcpy() or more complex but still essential ones like printf() would be missing. To remedy the situation many different options were evaluated and in the end it was settled on incorporating parts of “The OSKit Project” from the university of Utah [9]. The kit contains many parts of the standard libc, written either in a completely operating system agnostic way or having simple hooks that can be adapted to any new operating system.

Two of these hooks are the console_putchar() and console_putbytes() functions (see x86_lib.c). These are the low level routines used by all commands that work on the console, like printf(). In current Athomux development builds the result is output on serial port 1, which is a very convenient solution for debugging. Emulators like QEMU have the ability to redirect an emulated serial port to their standard console output, thus giving an easy and comfortable way to review the debugging messages. Optionally there is also code to send the output

to the screen, but it has been disabled as the user programs running under Athomux now need the screen real estate themselves.

Still some more work had to be put into the library as Athomux makes extensive use of long long data types in the printf() and sscanf() functions, which the OSKit libc did not support. This resulted in very strange and difficult to track down behavior in early builds of the native Athomux kernel as undetected problems or even stack corruptions deep within the innards of the Athomux brick support libraries occurred.

3.10.1 64-bit Maths Support

Not only the libc is missing, but the basic support functions from libgcc are also not available. These include the routines needed for doing long long 64-bit maths, especially the division and multiplication routines. As many data types in Athomux are of the 64-bit variety, this posed a bit of a problem.

The Linux kernel space version of Athomux, which also suffered under these limitations, remedied this by only allowing whole powers of 2 to be used as multiplicand/divisor, which can be done by pure bit-shifting. This restriction seemed too limiting for the native implementation, therefore the parts of the gcc3.3.6 source code that handle these types of operations have been surgically extracted and put into work in the file x86_math64.c.

4 Running and Debugging

Following is a quick introduction into running and debugging the current Athomux kernel, for anybody wanting to just try it out or further develop the existing code base.

The easiest solution is to use the pre-built floppy image provided along with source code. In this case the next section can be skipped, as it describes how to build a clean floppy or floppy image from scratch.

4.1 Installing GRUB

This can either be done by using a real floppy disc or, if the OS in question provides a loop back device, by using an image. Here the image approach is explained.

```
$ dd if=/dev/zero of=/tmp/floppy.img bs=512 count=2880
$ losetup /dev/loop0 /tmp/floppy.img
$ mke2fs /dev/loop0
$ mkdir -p /tmp/image
$ mount -t ext2 -o loop /dev/loop0 /tmp/image
$ mkdir -p /tmp/image/boot/grub
$ cp /boot/grub/stage1 /tmp/image/boot/grub
$ cp /boot/grub/stage2 /tmp/image/boot/grub
$ cp /boot/grub/e2fs_stage1_5 /tmp/image/boot/grub
$ grub
grub> device (fd0) /tmp/image/floppy.img
grub> root (fd0)
grub> setup (fd0)
grub> quit
```

Text 1: Installing GRUB

This creates a clean floppy image, formats it using the ext2 file system, mounts the image, copies the GRUB files onto it and installs the boot sector. Paths have to be adapted adequately, of course. What is missing is the grub configuration file menu.lst in /tmp/image/boot/grub. This could have the following example contents

```
# Boot automatically.
Timeout 0

# By default, boot the first entry.
Default 0

title  Athomux
root   (fd0)
kernel /kernel
```

Text 2: /boot/grub/menu.lst

4.2 Installing the Kernel

Quick turn-around times are essential for a quick and painless development process. This is achieved through small helper scripts which take over repeated tasks like installing the kernel and booting the system.

The following script **`installx86`** mounts the floppy image if necessary and copies the kernel into it. Normally it is called by the different other scripts that control the boot process and does not need to be called manually.

```
#!/bin/bash
if [ ! -e image/boot ]
then
      echo -n "Trying to mount floppy image... "
      modprobe loop
      mount -t ext2 -o sync,loop=/dev/loop0 floppy.img image
      echo "done"
fi
cp x86/x86/athomux_x86 image/kernel
```

Text 3: installx86 script

Note that the mount specifies the “sync” option. Without it, the floppy image file will not be immediately written to and an emulator using it afterwards will most likely not use the latest kernel version! The modprobe command may or may not be necessary on different systems.

4.3 Creating a real Boot Floppy

In order to try Athomux on a real PC system a boot floppy has to be created. This can simply be done by writing the image to a disc:

```
$ dd if=floppy.img of=/dev/fd0 bs=1024 conv=sync
$ sync
```

On Windows system the program rawrite[10] can be used for this purpose.

4.4 Booting the System with QEMU

The standard tool for developing the native variant of Athomux is the PC emulator QEMU. See also chapter 7 for the pros and cons of the different emulators.

When a sufficiently recent QEMU version is installed (development was done using version 0.8.2) then the simple **bootx86** script can be used to get a first taste of Athomux.

```
#!/bin/bash
./installx86
qemu -fda floppy.img -serial stdio -m 20 -boot a
```

Text 4: bootx86 script

The script tries to install the latest compiled kernel onto the floppy image (see chapter 4.2) and then tells QEMU to boot this image. All serial I/O is redirected to the standard console, so that the Athomux boot messages can be seen. Currently 20 megabytes is about the minimum to run the Athomux kernel.

The VGA output of the emulated PC can also be redirected to a virtual VNC session using the “-vnc” parameter , so that it can even be used over a network.

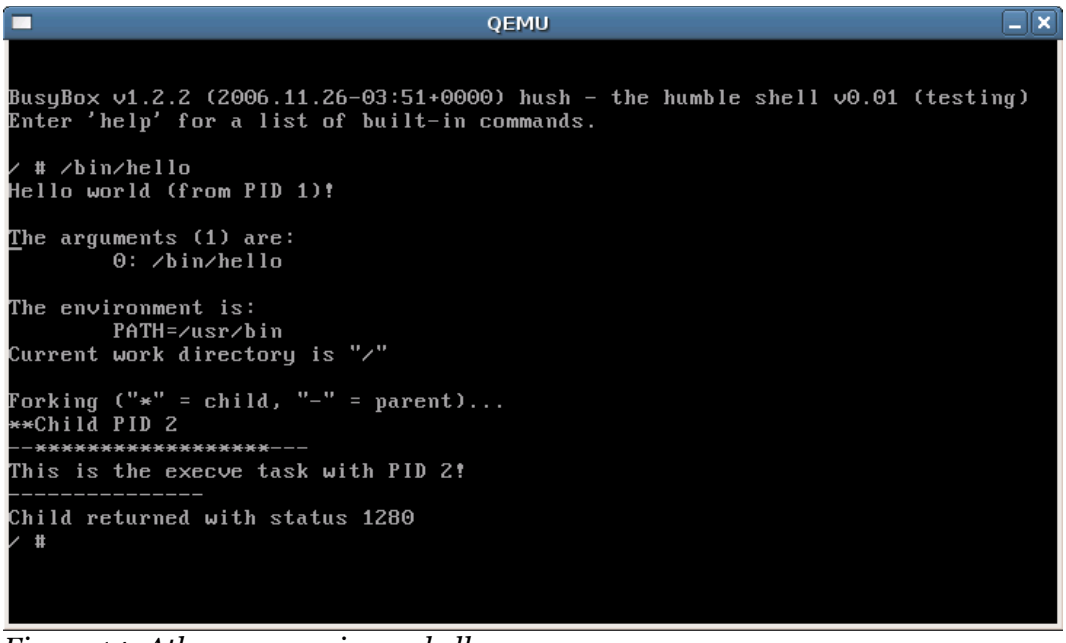

Figure 14: Athomux running a shell

The current Athomux build incorporates the Busybox[11] tools, which amongst others include a few simple command line shells. In the screen shot the “hush” shell is used to start the hello_world sample executable. This is a simple demonstration program that prints its command line arguments and environment, forks and in the fork executes itself with a special command line parameter that prevents it from recursion:

```
int main(int argc, char *argv[], char *envp[]) {
  if (argc > 1 && !strcmp(argv[1], "execve")) {
    printf("\nThis is the execve task with PID %d!\n", getpid());
    return 5;
  }
  printf("Hello world (from PID %d)!\n\n", getpid());
  int i;
  printf("The arguments (%i) are:\n", argc);
  for (i = 0; i < argc; i++) {
    printf("\t%i: %s\n", i, argv[i]);
  }
  printf("\nThe environment is:\n");
  i = 0;
  while (envp[i] != NULL)
    printf("\t%s\n", envp[i++]);

  char buffer[512];
  getcwd(buffer, 511);
  printf("Current work directory is \"%s\"\n", buffer);
  printf("\nForking (\"*\" = child, \"-\" = parent)...\n");
  pid_t pid = fork();
  if (!pid) {
    for (i = 0; i < 20; i++) {
      printf("*", i);
      fflush(stdout);
    }
    int res;
    char *exargv[3] = {"/init", "execve", 0};
    res = execve(argv[0], exargv, envp);
    printf("Execve returned %d\n", res);
    return 16;
  }
  printf("Child PID %i\n", pid);
  for (i = 0; i < 20; i++) {
    printf("-", i);
    fflush(stdout);
  }
  int status;
  wait4(pid, &status, 0, NULL);
  printf("\nChild returned with status %i\n", status);
  return 1;
}
```

Text 5: The hello_world example program as run by Athomux

The binary of the hello_world program is compiled using the dietlibc[6] library, but like BusyBox could also be compiled using uClibc[7]. It is notable that exactly the same binary that runs under Linux is employed in the Athomux tests.

Following is an excerpt of the debugging messages shown during system boot.

```
Athomux starting up...

kernel_code_base 0x100000, kernel_data_base 0x300000,
kernel_data_size 0xa00000
unused memory space base 0xd00000, size 0x1300000
paging tables set up...
memory initialised...
GDT set up
IDT set up
operation : on "map_dummy"
operation : on "adapt_meta"
error: OP map_dummy_out_0_getXgettranswait file map_dummy.ath
line 275 op(gettranswait) sect(0) mand(10): (!(_brick->all)-
>used[index]): illegal access at address 0
operation + on "cpu_x86"
operation : on "fs_simple"
operation : on "strategy_native"
Strategy brick starting up...
put file [/ROOT/bin/busybox]
error: OP map_dummy_out_0_getXgettranswait file map_dummy.ath
line 275 op(gettranswait) sect(1) mand(10): (!(_brick->all)-
>used[index]): illegal access at address 1000
error: OP dir_simple_out_0_TO_1_op file dir_simple.ath line 501
op(gettranswait) sect(1) mand(18): ((_args->log_addr) + (_args-
>log_len) > entry->off[(_args->sect_code)].off_len): bad region
access: 1000 > 0 (border=0)
error: OP dir_simple_out_0_TO_1_op file dir_simple.ath line 501
op(gettranswait) sect(1) mand(19): ((_args->log_addr) + (_args-
>log_len) > entry->off[(_args->sect_code)].off_len): bad region
access: 1000 > 0 (border=0)
put file [/ROOT/bin/lash]
put file [/ROOT/bin/hush]
put file [/ROOT/bin/hello]
+-----------------------------------+
| athomux: creating init process... |
+-----------------------------------+
=== create process [/ROOT/bin/hush], posix pid 0
static_elf_decoder instantiated
process_image instantiated
mmu_x86 instantiated
posix_x86 instantiated
char_device_kbd instantiated
kbd_us stdin instantiated
char_device_screen stdout instantiated
char_device_screen stderr instantiated
posix: 1
syscall_x86 pid: 0
posix: 2
mmu_x86: exec [/bin/hush], entry: [0x80480b0]
mmu: new task structure at 0xfa (index 1), len 250
Scheduler selected task 1/10001 (eip 0x80480b0, oldeip
0xc0128d27)
Page fault! Error code 0x4, address 0x80480b0, eip 0x80480b0
[...]
syscall eax=0x36! (0x0, 0x5401, 0x17fffe5c, 0x17fffea8, 0x0)
 | pid 0: ioctl fd 0, request 5401 [NYI]
exit to 0x8064b84, eax=0x0
[...]
```

Text 6: Athomux boot messages

4.5 Debugging

The Athomux kernel was exclusively developed remotely over a LAN or the Internet on a headless machine running Gentoo Linux. The connection was done using an SSH connection running the "GNU Screen" virtual terminal window manager. This fact is important as the debugging scripts developed need the Screen software to work.

The starting point is the **debugx86** script. It checks that the current terminal is running Screen, installs the latest compiled kernel into the floppy image, instantiates the **gdbstart** script on a separate Screen window and then runs QEMU with the "-s -S" options. These cause the emulator to activate its internal GDB stub and halt on the first instruction to be executed.

```
#!/bin/bash
if [ $TERM != screen ];
then
  echo "This script needs to be run within GNU Screen"
  exit
fi

./installx86
screen ./gdbstart
qemu -fda floppy.img -serial stdio -vnc 2 -m 20 -s -S
```

Text 7: The debugx86 script

In parallel the **gdbstart** script starts up. It waits a second to give QEMU a chance to boot up before starting GDB. As the Athomux kernel including the debugging symbols is too big for the floppy image, only a stripped binary is copied onto the image. To rectify this situation GDB is given the file with full debugging symbols as a parameter.

```
#!/bin/bash
# Let QEMU start remote session
sleep 1s
gdb -symbols=x86/x86/athomux_x86 -x gdb.remote
```

Text 8: The gdbstart script

Finally there is the gdb.remote file. It is executed by GDB before anything else and tells it to connect to the locally running GDB stub within QEMU and also sets up the source view mode for more convenience.

```
target remote localhost:1234
layout src
```

Text 9: The gdb.remote file

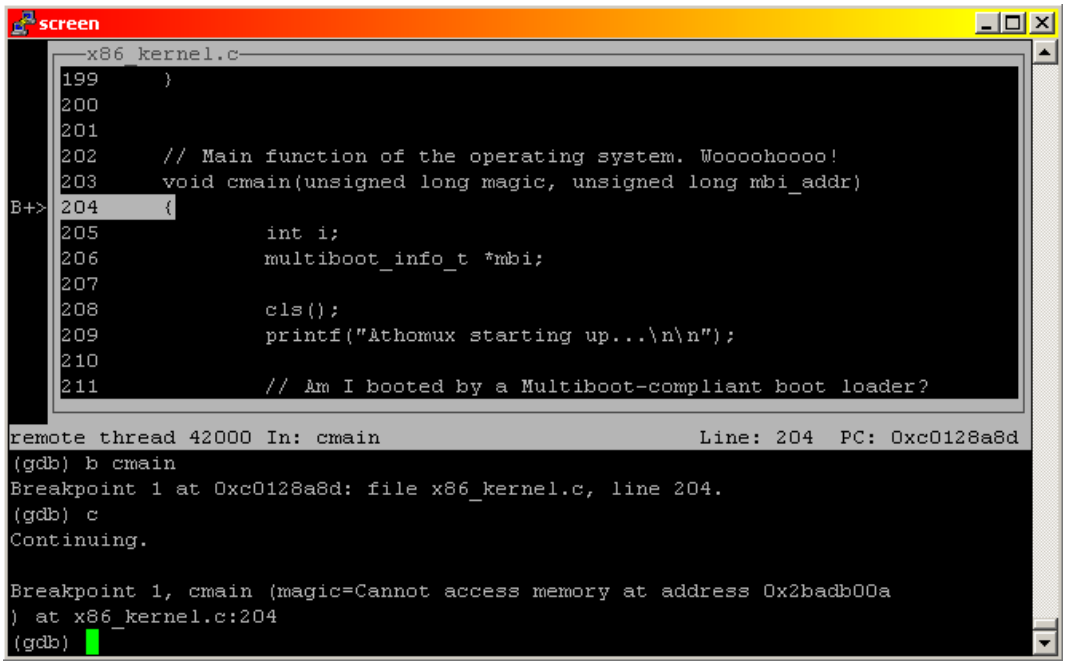

Figure 15: GDB debugging the Athomux kernel

From now on GDB can be used like in any other environment. All Athomux debugging symbols are available. Switching between the QEMU output and GDB can be done using the standard Screen keyboard shortcut “Ctrl+A a”.

5 Performance

It is a bit early to really benchmark the system, it is just too much in its infancy. But one test was done to at least get some rough idea of the magnitude of its performance. The test program was simple, it just prints halve a million stars to the standard output. This mainly benchmarks the system call interface and the driver system.

```
int main(int argc, char *argv[], char *envp[]) {
    int i;

    time_t t1 = time(NULL);
    for (i = 0; i < 500000; i++) {
        write(1, "*", 1);
    }
    time_t t2 = time(NULL);
    printf("\nTime1 %d\n", t1);
    printf("Time2 %d\n", t2);
    printf("Time taken: %d seconds\n", t2-t1);
    return 0;
}
```

Text 10: Benchmark source code

The contestants are:

1. **Athomux** with disabled kernel debugging messages
2. **Alcolix** mini Linux distribution using kernel 2.4.20
3. **Damn Small Linux** distribution using kernel 2.4.26
4. **Knoppix** distribution using kernel 2.4.20

The systems were all run in a VirtualPC session on a Pentium III 850MHz PC running Windows 2000. All systems were told to just boot into text mode and all ran exactly the same benchmark binary. The test was executed 5 times and the results are given in seconds.

Athomux	*Alcolix*	*DSL*	*Knoppix*
67	86	41	42
68	88	41	41
69	88	41	42
67	87	40	41
68	86	41	41

As can be seen all values are pretty consistent. The Knoppix and the Damn Small Linux system behave almost the same and share the first place. Alcolix gets

last place but it did seem to change into a different text mode, which might explain the discrepancy. Athomux, being totally unoptimized, does not fall too much behind the two much more refined but also more feature-full winners.

All in all one can deduce that the Athomux approach is not inherently slower than other systems.

6 Conclusions

6.1 Summary

The components that were to be developed over the course of this thesis, mainly the *cpu_x86* and *mmu_x86* bricks along with all the underlying boot, C library and development infrastructure, have been completed and, despite the high complexity and very difficult debugging situations involved in creating a new operating system *from scratch*, work very reliably. Even some advanced virtual memory techniques like “copy on write” have been successfully implemented. All in all it can be said that while the theory of things is often simple enough, reality with actual hardware isn't, and most time of this work was spend working on reality.

The POSIX system on top of all that is also working to a degree that even a simple command shell plus applications can be run.

Methods have been devised and tools have been identified that make further developments on the standalone system much easier.

Of course one single thesis is far from enough to write a new operating system that can compete with anything already on the market. But this was not the goal of this work. Instead it has been shown that a standalone Athomux version is possible and that its performance does not inherently suffer from the brick and nest approach.

6.2 Problems

One thing can surely be said after all this work: modular kernels are hard to do. The structures are always good in theory and look great in diagrams, but in the real world structures are often much more complex than can be seen at first sight. In a traditional monolithic kernel referring to some functionality in another module is mostly pretty easy: include its header file at the top and you can use its functions. Deal done. Of course this can lead to a decrease in locality and

eventually in a maintenance nightmare, but it is definitely a fast and easy way to program.

The connection based approach on the other hand can be very hard. Often one develops something in brick X and then comes to a point where one needs some tiny piece of information from brick Y. Brick Y of course is at least 3 steps away and there is basically no clean and easy way to get access to it, other than modifying all bricks in between to pass the address information along. This problem could be solved by registering important bricks like *cpu_x86* at a central name space so that every brick can connect to it without much hassle, in essence creating a more shallow structure instead of a deep tree. The Athomux philosophy certainly can handle both approaches.

Another problem is the compiling and debugging of Athomux code. One missing bracket can cause the Athomux preprocessor to throw in the towel without much information on where the actual problem is. And when the preprocessor lets the file pass but GCC finds a problem, the line number in the error message has no resemblance whatsoever with the code written in the Athomux language. Similarly the debugging with GDB does not happen in the written Athomux code, but in the code output by the preprocessor. All these points do not make development impossible, but simply more difficult than in other environments.

6.3 Future Work

There are literally a million places where the existing code can be extended and optimized. The kernel code quality itself is remarkably mature but more work has to be put into extending its functionality and improving the POSIX system that runs on top of it. Or devising a different process model other than POSIX altogether.

As an example for improvement, the kernel memory handling should probably be cleaned up. Currently there is a fixed memory pool for serving the malloc needs of the existing code. Using the malloc function is however against the Athomux philosophy and when all uses have eventually been eliminated this part of the the kernel can be cleaned up accordingly.

In any case, the work described herein is more an intermediate step between a monolithic kernel and the real Athomux philosophy. But with this important step working, future iterations can be gradually done to move away from the traditional approach of kernel writing to the new completely component based approach.

Last but not least, while all basic concepts of MMU usage and paging are already implemented, swapping of memory to a swap hard disc or partition is not yet done. The memory needs of all tasks currently cannot exceed the physical memory space. Like *sched_x86* a separate strategy brick that decides which pages to swap out to disc could be devised, so that different strategies can easily be tested or even exchanged at runtime.

7 Appendix: Emulators

Emulators are very important tools when it comes to developing an operating system. Several have been tried during the creation of the Athomux kernel and following is a short description of the pros and contras for the main choices.

7.1 Bochs

Bochs (pronounced “box”) is one of the oldest free PC emulators around. It completely emulates the whole PC hardware, including the CPU (interpreted), even when running on an x86 compatible processor. This has both the advantage that every little bit of CPU state is available within the emulator and the disadvantage that the emulation is very slow. It has an integrated debugger, which in itself has a huge potential as it has access to all internal state, but even in the most current version (2.3) it is so poorly done as to be nearly unusable. On the plus side Bochs includes plenty of debugging output and error messages if something goes wrong, but the flawed debugger plus its inherent slowness still makes Bochs a poor choice for Athomux development.

As the source code is available specially instrumented builds can be done which can be useful for those very hard to find bugs in the deepest innards of the operating system or hardware drivers. However, as the lowest levels of the Athomux kernel have become more and more mature this is less necessary.

7.2 QEMU

Unlike Bochs, QEMU uses a just-in-time compiler for its CPU emulation and when the host machine possesses an x86 compatible processor it even includes a special mode that directly employs this circumstance for maximum emulation speed. As a third and fastest option there is a kernel mode version for Linux which runs the code natively as long as possible and only virtualizes the privileged parts. This has however not been tested.

QEMU does not feature any inbuilt debugger but instead includes a GDB stub, so that a normal GDB debugger can be attached remotely via TCP/IP. This way even source level debugging is possible, which makes this combination (QEMU+GDB) one of the most powerful tools for Athomux development (see chapter 4.5 for more details on this).

The only main disadvantage is that GDB is a pretty poor debugger when it comes to debugging raw assembly code and it cannot handle segmented memory at all. But as the latest Athomux builds now incorporate a real flat memory model the latter point is not as important anymore as it used to be.

Like Bochs, QEMU is free and the source code is available, too. Therefore specially instrumented builds can be done if the GDB debugging functionality is not enough.

An emulated serial port can be redirected to the standard console so that all kernel debugging messages are easily available. This is actually one of the most important options of all.

7.3 VirtualPC

Microsoft's VirtualPC is one of the fastest options when it comes to emulation of a full PC, but except for the occasional test run its use in development is limited. It does not feature any integrated debugging facilities, the only option would be to built a small debugging stub into Athomux itself. It does however provide the option to redirect an emulated serial port into a file, so that at least the Athomux debugging messages are available for review.

VirtualPC is freely available, but not open-source. It is limited to Windows as a host system.

7.4 VMWare

VMWare basically has the same properties and limitations as VirtualPC. Only advantage is that it also runs on top of Linux. Some versions of VMWare are freely available, too.

List of References

1: **Schöbel-Theuer, Thomas:** *Eine neue Architektur für Betriebssysteme,* 2005
Unpublished manuscript of a monography, available on request from ts@athomux.net

2: **Niebling, Florian:** *Development of a Process Model for Athomux,* 2004
http://www.athomux.net/papers/diplomarbeit_flo.pdf

3: **Kilgus, Marcel:** *QEMU developer mailing list,* 2006
http://www.mail-archive.com/qemu-devel@nongnu.org/msg07362.html

4: **Shukla, Vikram:** *Explore the Linux memory model,* 2006
http://www-128.ibm.com/developerworks/linux/library/l-memmod/

5: **Lea, Doug:** *Free malloc implementation,* 2005
ftp://g.oswego.edu/pub/misc/

6: **von Leitner, Felix:** *diet libc - a libc optimized for small size,* 2006
http://www.fefe.de/dietlibc/

7: **Andersen, Erik et al:** *uClibc - A C library for embedded Linux*, 2006
http://uclibc.org/

8: **Recktenwald, Hans-Peter:** *Reference to Linux 2.{2,4,6} System Calls for Assembly Level Access*, 2004
http://www.lxhp.in-berlin.de/lhpsyscal.html

9: **University of Utah:** *The OSKit project,* 2002
http://www.cs.utah.edu/flux/oskit/

10: **Newbigin, John:** *RawWrite image writer*, 2006
http://www.chrysocome.net/rawwrite

11: **Landley, Rob et al:** *BusyBox: The Swiss Army Knife of Embedded Linux,* 2006
http://www.busybox.net/

Declaration

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

_____

(Marcel Kilgus)