Acaros design and implementation

Marko Mikuličić ...

April 12, 2004

Contents

1 Introduction **3**
- 1.1 Design notes . . . 3
- 1.2 Virtual memory . . . 4
- 1.3 Bootstrap . . . 5
- 1.4 Documentation . . . 6
- 1.5 History . . . 6

2 Build system **7**
- 2.1 Introduction . . . 7
- 2.2 Version control . . . 7
- 2.3 Makefiles . . . 9
- 2.4 Configuration . . . 9
- 2.5 Build . . . 9
- 2.6 Installation . . . 10
  - 2.6.1 Emulators . . . 10
  - 2.6.2 Booting Acaros . . . 10
- 2.7 Debugging . . . 11
- 2.8 Documentation . . . 12
- 2.9 Implementation . . . 12

3 Loader **13**

4 Kernel **15**
- 4.1 Introduction . . . 15
- 4.2 Building . . . 15
- 4.3 Boot . . . 15
- 4.4 KDB . . . 15
  - 4.4.1 Interruption, resumption . . . 16
  - 4.4.2 Program state . . . 16
  - 4.4.3 Tricks & details . . . 17
  - 4.4.4 Performance issues . . . 17

5 Memory Manager **19**
5.1 Physical memory allocation 19
5.2 Virtual memory mapping 19
5.3 Pools 20
5.4 Virtual memory range allocator 21
5.4.1 Free list allocator 21
5.4.2 Binary buddy allocator 21
5.5 Initialization 21
5.6 Platform indepency 22
5.7 Page tables 22

6 IO **25**

A Library helpers **27**
A.1 libstand 27
A.2 libalgo 27
A.2.1 Linked lists 27
A.2.2 AVL tree 29

B Coding Style **31**
B.1 Module names 31
B.2 Function names 31
B.3 Type names 31

List of Figures

A.1 Linked list example. 27
A.2 Linked list ordering. 28
A.3 List traversal. 28

Chapter 1

Introduction

Acaros is an operating system kernel targeted for workstation and server class machines. The design permits easy ports to embedded class machines but this document will focus only on the implementation on mid/high range machines which include MMU (virtual memory), privilege separation (kernel/user), external bus (PCI, ...), DMA, and programmable interrupt controllers.

Acaros will be the kernel of the Kaos project which will build an complete system based on Acaros .

1.1 Design notes

Acaros does not follow any schoolbook design. It is not a microkernel design but neither a monolithic UNIX style kernel. Acaros design goal is to merge microkernel concepts with performance needs allowing processes to run inside kernel space or user space using basically the same API, thus allowing easier developing and debugging in user space and increased security for untested drivers or extensions[1], while maintaining breakthrough performance for critical path code like server networking and filesystem operation.

The design allows an administrator (or an end user though the installer program) to choose the level of protection vs. performance, moving in and out modules from user to kernel.

As stated before this is the design goal, not necessarly the goal for the first implementation, but many design decisions are influenced by that.

Other design decision is to clearly protocolize intra-module and intra-layer communication reducing the binary incompatibility between different releases/builds [2]. Object oriented message passing is choosen to implement this abstraction.

[1] expecially third party

[2] Like linux, most notably

We have developed a very simple and limited object oriented extension to the C language: CO[3]. It is similar to Objective C but with less emphasis on smalltalk emulation and with far simpler implementation and with less features. The lack of advanced OO programming techniques available to the kernel programmer should help him avoid the abuse of OO in time critical applications such the kernel and keep clear the releationship between objects, methods and backing memory. The CO compiler[4] is implemented as a preprocessor to C.

No existing object oriented programming language was well suited for kernel core development, partially because it we found difficulties porting the language runtime without basic OS support and also we would loose compatibility with the mainstream compiler at each new version. We didn't want force users to use a specific version of the C compiler nor special build tools or compilers. For now we only assume the GNU tool chain and use some of its peculiarities but it the future a more compiler agnostic approach will be tried.[5]

GNU C was choosen for infinite reasions. Portability is granted by the widespread use of the GNU C Compiler and its infinite incarnations for practically every known CPU. Explicit use of GNU C extensions was choosen above macros with compiler specific code to avoid preprocessor junk and improve readability.

1.2 Virtual memory

Acaros kernel uses paged virtual memory for all memory management and does not direct map physical memory as other OS do[6].

The virtual memory management is conceptually similar to those from Windows™ NT and Solaris™, but is not meant to be compatible with them.

Acaros is a modern OS and thus implements the common memory management functinos like memory sharing, copy-on-write, memory mapped files, device memory mapping, demand paging, shared mem IPC and zero copy IO (networking).

Platform dependent page structure handling is done throgh a dedicated HAT (Hardware Address Translation) layer, part of the MAL (Machine Ab-

[3] The acronym is open to free interpretation, from *C object* to *Che cazzO*

[4] coc

[5] Is not so easy to be compiler agnostic without the use of too much conditional compilation...

[6] Linux maps all physical memory in the kernel address space. The consequence of this design is that the amount of physical memory is restricted to 1GB/2GB/2GB (depends on kernel configuration: more address space to user mode processes, less physical memory available). Recent patches relax this limitation at cost of performance and code readability.

straction Layer) [7]. Platform independent code does not access directly page structures through macros like linux for example, thus it is not necessary to emulate a three level page structure like it does.

1.3 Bootstrap

The execution of Acaros is divided basically in two parts:

1. loading kernel
2. executing kernel

it seems obvious but it depends on what we mean for "loading" the kernel:

The kernel itself uses memory resources for its code, data and stack but the memory management (and paging) is not yet initialized when the kernel is first run. One approach would be to put a careful written assembly and C code which prepares memory management structures and initialize paging, in order to pass the execution to the kernel once its living in the right address space[8].

Acaros instead uses an intelligent loader which loads the kernel at the right virtual addresses contained in its executable image[9], like any other program, initializing paging and keeping track of each physical and virtual memory allocation and passing a structure describing the memory layout to the kernel. The kernel can then go on and still use loader memory management functions for initial memory allocation (like allocation of the Page Frame Database) and then use this information to fill its memory management structures to reflect the initial allocation. After that the initial the full featured memory management routines can be used.

This strategy avoid knowing the layout of the kernel memory management structures when initializing the machine and improves development rates and platform independecy, at the cost, however, of a little more work and some mental confusion.

The loader itself is platform dependent but has a common protocol with the kernel entry point and the kernel memory manager subsystem. It is like a little OS with limited knowledge of memory management and executable loading. Eventually the loader stage could also be able to load the kernel image and inital modules from disk or it could delegate this to stage 0 loaders like GRUB. Every architecture has it's separation of tasks, involving potentially other entities interacting before the kernel itself boots, for example on the PC architecture the loader interacts with the a multiboot

[7] from an latin stem which means *evil* with an italian acception of *pain*, which is very well suited in hardware related stuff...

[8] More information about that at **??**

[9] Currently ELF is choosen

compliant boot loader[10] which is responsable for loading the loader and the kernel from disk or network, and switching to protected mode [11]

The kernel is a normal ELF executable image compatible with the native Linux executable format, but probably it should work with solaris build tools as well.

1.4 Documentation

Other than this book there is also a reference documentation generated with doxygen in src/doc/api/html and src/doc/api/latex. See building system at page 7.

1.5 History

The project begun when in late 2002 my professor of *technical computer science* at the SUPSI[12] showed us a little embedded real time OS called uCos™, which came you needed came with a crappy build system and you needed to patch several files in order to compile, it ran with a 16 bit turbo C compiler and it's scheduler didn't allow two threads to have the same priority.

While it is surely a good choice in many application, and it's pors to many platforms are certainly useful I really don't understand why I or my university should pay royalities to a thing like this. So I wrote a parody of uCos (spelled micro C O.S) and called it mikuCos (spelled m i q c o s), which quickly had a full blown time round robin scheduler, keyboard, timer, syncronization primitives, and then arrived a vt100 emulator, uart, pc speaker, virtual filesystem, pci, network driver, paging. The code was so ugly and eventually was so filled with various memory corruptions that it was wiser to rewrite everything with less hurry, and so Acaros was born.

It started up as a joke mainly because I've found out how simple it was to load a ELF executable directly from the GRUB boot loader using standard tools and with little or no assembler, so the I skipped the most annoying part, the bootloader itself, on which people usually get so frustrated that they loose interest in OS writing (or simply loose their time) ...

[10] Like grub

[11] this way simulating finally a real 32-bit machine. Acaros has no 16-bit code, not even on bootstrap ! :-)

[12] switzerland university of applied sciences

Chapter 2

Build system

2.1 Introduction

Acaros attempts to have a clean, easy to use and *template free* build system. That means that unlike *autoconf/automake*-like systems, the Makefiles are not dynamically generated, but instead they are a handful of very short and modular makefile scripts which are included from a 3-4 line makefile in every directory.

The makefile in most directories need not to be changed, even when adding source files, so simplifying the version control and branch merging.

2.2 Version control

Subversion[1] was chose to host the project, because ... you will find out using it.

Repository access requires SSH access for now. The repository path is:

```
svn+ssh://sottosale.net/home/users/kaos/repos/acaros/trunk/
```

Basic subversion commands description assumes the REPOS to be set to the value of the repository path shown above.

Checking out

```
svn co $REPOS acaros
```

This will create a subdirectory called acaros the current directory. Obviously you could named as you like.

This directory will be a *working dir*.

[1] http://svnbook.red-bean.com/svnbook/

Updating

```
svn update
svn up
```

Will fetch new releases from the server.

Getting status

```
svn status
svn st
```

Will list in what state the files in the in the working dir in respect of the lastly updated revision in the repository"

? not versioned

! missing

M locally modified

D locally deleted

A locally added

L locked

...

Adding files and directories

```
svn add (file|dir)*
svn add myfile.c mydir
```

Renaming files and directories

```
svn mv oldName newName
svn mv blabla/dir1 there/dirzz
```

Major feature. This feature is the most important improvement over CVS.

Ignore some files

```
svn propedit svn:ignore dir1
svn propget svn:ignore dir1 |
  svn propset svn:ignore -F /dev/stdin dir2
```

Allows to specify filters (like *.a) to compilation products or other files that you don't want to be versioned (or to produce the annoying "?" in svn status)

2.3 Makefiles

Most makefiles (except src/Makefile, src/kernel/Makefile, ...) are very short and duplicate the bare minimum.

The only part which changes is list of subdirectories which will be recursively traversen in the build phase.

This subdirectories are called *subsystems* and defined in the SUBSYSTEM-y variable. Makefiles need only to append the named of the subdirectories that should be processed according to the current configuration. This is done using the following trick:

```
SUBSYSTEMS-y += log mm
SUBSYSTEMS-$(CONFIG_KDB) += kdb
```

In this example the subsystems `log` and `mm` are forcefully enable while the subsystem `kdb` will be enabled only if the variable `CONFIG_KDB` evaulates to "y".[2]

2.4 Configuration

The Acaros compilation can be tailored with optional parameters using a ncurses menu driven interface, using a simple `make config` command.

The program that handles this configuration was "stolen" from the linux build system.

It basically works parsing the Kconfig files and generating a .config and an include/autoconf.h file. .config will be included in every makefile while autoconf.h will be include by source files if the implement some feature conditionally.

Makefiles can conditionally include subdirectories in the subsystem path for the module; this way for example the kernel debugger will not even be compiled if you don't enable it.

2.5 Build

A simple `make` invocation should build the whole kernel according to current configuration.

The default make target doesn't include a dependency tree generation because it's slow and is left to manual use. So unless you don't issue a `make deps` modifying a header file will not recompile the dependent C files.

Some times, when the dependency is not enough and a complete rebuild is needed, a handy command `make rebuild` does the job for you, a complete clean,deps,make,install pass.

[2] defined in .config if enabled from menu

The build system build the two main components of the system leaving them at their original locations (arch/$ARCH/loader/loader and kernel/kernel kernel/acaros.tar.gz. The kernel image is encapsulated in a tar archive with other boot modules and drivers. On boot this tar archive is opened by the loader which locates the kernel and loads it.[3]

2.6 Installation

Since Acaros development cycle is intended to be fast and with extensive testing of every line of code[4], there must be a way to quikly install and boot the target system.

2.6.1 Emulators

One way to quickly test Acaros is to use an emulator like bochs or vmware,

Vmware

Works. Network + vesa too.

The build system assumes you have your virtual machine in /vmware/acaros [5], and it's important for serial port redirection, used mostly by the kernel debugger.

Important vmware configuration are:

Memory choose whatever you want. Acaros assumes ¿1.5, but just for now.

Bochs

Works but hangs on kernel debugger.

The .bochsrc file is provided.

It boots only using the floppy image.

qemu

It doesn't work with Acaros ...

2.6.2 Booting Acaros

There are basically two ways quickly prepare to boot Acaros :

- floppy images

[3] This way the multiboot command line can stay short.

[4] a sort of extreme programming but less extremist

[5] Actually instead of acaros it searches the current branch name, that is the name of the toplevel directory (the one below src).

- network booting

`make install` will automatically reckognize which boot system is enabled [6] and install the kernel in the right places.

This magic needs some collaboration however, as explained in the next sections:

Floppy image

You need a floppy image with grub installed on it. You can download one from http://www.acari.org/download/acaros.img

Once you put this file in the root of the build system you can now choose to mount it manually or using the `make loop_mount` command which exploits the `sudo` command, and mounts the image in loopback for you.

The image should be mounted in the `image` directory so that `make install` can find it.

The `support` directory contains an example grub configuration file.

Network boot

`make install` searches a /`home/tftp` directory and if it exists it copies the kernel and the loader in it.

Since grub is needed to boot the kernel you need to compile `pxegrub` and configure `dhcp` and `in.tftpd` to run it for the MAC associated with your testing machine.[7]

The `support` directory contains an example grub configuration file for network boot.

2.7 Debugging

The kernel is by default compiled with all debugging symbols enabled.

The build system helps also debugging by implementing a `make debug` command which installs the kernel, spawns a *gdbredirector* and a gdb instance.

You need only to attach to the tcp port 1235, if you use VMware, with:

```
target remote :1235
```

or directly to the serial port

```
target remote /dev/ttyS0
```

[6] By looking if the floppy image is mounted, or a tftp directory is present

[7] real or emulated (vmware can boot from pxe as of version 4.5)

and you will attach a running kernel through the serial port.

VMware can bind an emulated serial port to an unix domain socket (which he calls incorrectly *pipe*). The `gdbredirector` program converts TCP/IP from gdb to unix domain socket on the vmware side.[8]

2.8 Documentation

This documentation is located in src/doc and is builded with a simple make. The reference manual is build using make api, and it outputs api/html and api/latex. If you want to generate the pdf reference manual you should do another make inside src/doc/api/latex.

2.9 Implementation

TODO ...

[8] It can also be used for analyzing the gdb protocol

Chapter 3

Loader

Chapter 4

Kernel

4.1 Introduction

4.2 Building

The kernel is built as a normal ELF static executable without standard libraries. A linux compiler can produce suitable format and code.

The kernel is linked to run in the upper virtual memory area (should be configurable but now at 0xe0000000, 3.5 GB). The loader reads the addresses in the ELF header (which can be inspected with objdump and allocates and copies the kernel text and data segments in the allocated virtual memory regions.

4.3 Boot

The kernel, when executed, runs directly with virtual memory enabled.[1]

The kernel entry point is it's _start function. The loader passes a structure that describes the arguments, the physical and virtual memory maps, the loaded modules and other things.[2]

Then vital builtin modules are initialized (mal, kdb[3], mm, co, drivers[4]) and then the kernels friendly panics.

That's all folks :-)

4.4 KDB

The kernel debugger is a little and tricky part of the kernel. It implements the gdb serial protocol and implements stopping, resuming, breakpointing,

[1] Set up by the loader or on some architecture by the boot firmware
[2] Vesa informations, for example on PCs.
[3] Optional.
[4] in ultrabug branch

single stepping, memory and register reading and writing.

Using this simple primitives gdb can implement full source level debugging.

I/O through the serial is handled using a builting driver inside the kdb in order to not depend on the rest of the kernel. I/O is interrupt driven and character based.

See the info page for gdb under "* Remote Protocol::" for detailed description of the protocol.

One of the design rules of kdb was that it must use a fixed amount of memory and be able to handle big requests without allocating memory for buffers.

So it was implemented using a state machine that accept characters at imput and depending on the character changes the state. This is very good for parsing the packet, parsing hexadecimal addresses but showed to be quite difficult to maintain design rule if gdb askes us to write big amounts of memory, and passes a big stream of data, and of course if gdb doesn't ask us to do so, then what's about implementing it as a state machine at all? pure mental masturbation?

4.4.1 Interruption, resumption

Since the reception of characters is interrupt driven we can interrupt the kernel whatever it is doing if we got a special condition:

1. First time the debugger sent a valid packet (debugger is attacching)
2. Debugger sent character 0x3 (INT)
3. A breakpoint exception occour (int 3)
4. A single step condition occour (int 1)

In order to block the kernel indefinitely in this cases, kdb simply never returns from the interrupt handler but spins on a global variable waiting it to change state.

Resetting this variable the interrupted kernel will continue to execute as nothing happens because the serial port interrupt or breakpoint interrupt has saved all the state and will restore it upon return.

This loop can be interrupted only by other serial port interrupts which eventually can resume the execution later on.

4.4.2 Program state

Gdb needs to know the current state of the machine registers every time the target stops.

Kdb can send the content of the registers "on demand" responding to the "G" gdb command or it can feed gdb with the information of the most important registers directly when reporting that software is stopped, with the "T" packet (see gdb manual).

In any case it must access the state saved on interrupt. mal handles this very conveniently using a short assembler wrapper in front of every interrupt handler which pushes on the stack all the registers so that you can access them from C using mal_trapFrame (on x86). Modifing this registers has a real effect of modifying the registers of the real program once resumed. For example single step is enabled updating the saved eflags register image setting the trace bit.

4.4.3 Tricks & details

When executing an breakpoint instruction, on resume the CPU will issue again the same instruction. This is correct, because normally gdb will handle this situation deleting the breakpoint and resetting it later, but if the breakpoint is generated inside the kernel, for example by a panic or manual debugging, before returning from the breakpoint interrupt we must check whether the breakpoint instruction is still there and if it is it means that gdb has not set this break point, so we must skip it. (incrementing the instruction pointer)

4.4.4 Performance issues

TODO

Reading and writing is done using byte by byte page protection checking and this could be optimized

Chapter 5

Memory Manager

The Memory Manger handles memory pages for kernel and user processes. The allocation of variable sized memory regions in pools (heaps) are handled by specific Memory Manager subsystems (Bay Watch, and Slab).

5.1 Physical memory allocation

The Memory Manager maintains a database containing a structure for each physical page in memory. This structure is called Page Frame Database.

The physical memory is divided in segments of contiguous pages. Physical memory segments are maintained in AVL tree rooted in the PFD[1]. Each segment contains the start and end adresses and an array of page_frame stucture which contains state information for each page in the range.[2]

The PFD contains the list heads for several lists (used, free, bad, modified, ...) which group together pages frames with the same state. Since a page frame cannot have several states at the same time, the page frame structure contains a link field and it is an entry in a doubly linked list.

Allocation of free pages is simple as getting the next page frame on the free_list.

It will be considered in the future to move to a binary buddy algorithm, in order to allocate efficently contigous blocks of pages. However it should not be a big performance problem, but eventually it can be an issue with device drivers requiring contigous regions bigger than one page...

5.2 Virtual memory mapping

Virtual memory mapping are handled though address_space structures which completely define a virtual memory layout of a process (the kernel also has an

[1] Page Frame Database

[2] The current implementation uses a big array for the whole memory and puts non RAM memory in a "bad" list.

adress space). Each adress space contains a set of region objects containing the starting address, the size and common protection information. The region objects have methods for handling page-in and page-out actions and provide generic interface between various memory types.

For example: memory mapped exectable code page could be simply thrown away because it can be readed from disk but a modified data page should be saved in the page file [3]. region types can be implemented subclassing and customizing the Region class or simply implementing an object with the same protocol[4].

Regions are maintaned in a AVL tree ordered by their address range so that a range containing a particular address can be found very quickly.

5.3 Pools

In some occasions we want to be able to allocate portions of a address space region and be able to track which areas are free and which are used.

For example the slab allocator allocates blocks of 2^n pages from virtual memory.

Virtual memory regions' ranges themselves are allocated from the whole kernel address space, to avoid overlapping.[5]

The virtual memory range allocation is done only on page basis, for smaller blocks the slab allocator or other general purpose allocators are used.

The main characteristic for this allocator is that unlike most other allocators can't keep free list informations sparse in the virtual space itself because free pages may (and should) be unmapped. If the allocator needs to use free space to contain metadata it should use most of of a page once it allocates it as a metadata store. (see **??**)

It's mostly used as a page provider of the slab allocator, although it can be used directly.

OOP The range allocator is implemented in object oriended style, with different subclasses implementing different policies.

The area(s) on with an allocator operates is called a pool and the area used to maintain the metadata is called a pool control.

If the pool control is located inside the pool itself it's called a resident pool control.

[3] usually called swap in unix jargon

[4] See CO language, **??**

[5] Currently a sequential scan of reserved regions is done and the first address that matches is accepted. Not free list handling.

Global pool The global pool is a pool containing the whole virtual memory range. It's used primary for allocating memory regions and, by extension, all other pools.

By definition the global pool has a resident pool control. The metadata must be placed in a partucular zone to avoid conflicts in memory layout policy. For example it cannot be placed in the user space or in hyperspace, ...

5.4 Virtual memory range allocator

Two main algorithms are implementd:

- free list allocator
- binary buddy allocator

5.4.1 Free list allocator

The classical free list allocator uses free space itself to contain link fields used to link free space together. The virtual memory range allocator cannot waste whole pages just to maintain link to the next free range. Also used space header cannot be placed before actual data for the same reasion.

Metadata is allocated inside the region in paged sized chunks linked together to form a chain. Each chain contains a header and a number of equal sized slots containing metadata itself. The header contains a list head that links the chunks, a pointer to the first free slot, and a free slot count.

Slots are allocated using a simple free list algorithm. Free slots are linked together and used slots contain metadata.

When no more free slots can be found a new chunk is allocated [6] and added to the list. When a chunk returns to have all free slots, the chunk is deallocated and removed from the chunk list.

5.4.2 Binary buddy allocator

The binary buddy allocator (not yet implemented) must use a non resident pool control.

5.5 Initialization

The loader provides a list of virtual memory areas with a string tag describing why they are mapped and what they contain. The loader also gives us

[6] Actually a free slot is kept for allocating this new chunk

a list of areas of allocated physical memory pages, and a complete memory map indicating which memory ranges are mapped to RAM and what is device or ROM memory.

The Memory Manger allocates the PFD memory using for the last time loader functions and then translates the primitive loader maps into PFD and address space structures. It uses the libstand physical memory map to construct physical memory segments, and uses virtual memory mappings to remove them from the free_list and add them to the used_list.

When initialization is done Memory Management methods are used to initialize other kernel subsystems.

5.6 Platform indepency

In order to be platform independed the Memory Manager subsystem has a platform specific layer (HAT, Hardware Translation Layer) which hides platform specific issues from generic code. However is should be first clear what is common bewteen platforms and what not.

Most hardware platforms implement an unique address space and the system splits it in user and kernel areas[7], and it may seem reasonable to abstract this split in generic code but there is at least one platform that allows completly separate user and kernel spaces (UltraSparc).

Since each hardware address space has a number, each element in virtual memory can be identified with a pair (hardware address space / address). For platforms which don't support harware implemented address spaces software assigns address space numbers to user and kernel spaces.

The kernel maintains an array of adress spaces indexed with the address space identifiers. The HAT layer generates the pair (ASI/address) using hardware ASI or software ASI. This layer of indirection does not cost much because the kernel would have to maintain separate address space trees anyways, because user contexts must be changed at every context switch and kernel address space context should remain unchanged.

5.7 Page tables

This discussion is a bit x86 centric.. but should apply most standard paging models, i.e the ones that use hardware page table traversal...

Page tables are structures contained in memory. The MMU knows the physical address of the main page table[8], and all pointers in the page tables are physical pointers.

[7] Usually in 2G/2G or 3G/1G splits

[8] Page directory, in x86 terminology.

Kernel itself runs in virtual memory. Initial page tables are constructed by the loader before activating paging. There are mappings for the kernel code, data and stack.

The difficulty arises when the kernel tries to access the page tables since it should map them in virtual memory before accessing it.

There are basically two solutions:

- Map all physical memory to a fixed place in virtual space
- Make the page directory be itself a page table for a special region called hyperspace

The first solution would appear good when considering machines with few physical memory and a 2GB/2GB user/kernel split, but today mid-range machines can have considerably more RAM than it's allowed to map in the restricted kernel address space. Obviously this consideration is valid mostly considering the x86 architecture which has only a 32-bit address space and lacks of hardware address space context (requiring sharing the limited 32-bit address space). Linux suffers from this design limitation and for beeing able to use *high memory*[9] one must compile the kernel with a particular option and inside the kernel it is very tricky to handle.

The second solution is to map the page tables themselves in virtual memory. This is done simply by pointing a page directory entry back to the page directory itself so that a virtual address inside the range associated with that PDE is looked up using the page directory as a normal page table, thus accessing all page tables as if they were normal pages inside this *hyperspace*.

Moreover if a page table is not present a normal page fault is generated so that we can implement on demand paging for page tables.

Location The hyperspace is located currently at the last 4MB of address space, but it can be considered moving it to the border of user and kernel space, in such position that the user pages are mapped in hyperspace below the user/kernel limit and the kernel pages above it. However I don't currently see any practical reasion of doing so other than elegance.

[9] Memory above the range (usually 900MB) which can still be directly mapped in kernel space

Chapter 6

IO

Appendix A

Library helpers

Acaros supplies a set of useful library functions and data structures.

A.1 libstand

A.2 libalgo

A.2.1 Linked lists

The doubly linked list implementation used in acaros is derived from the linux kernel sources. It is designed for high performance and low overhead. Instead of manage separate *nodes*, linking information is embedded in the manged objects.

Example A.1 shows how to create a structure that can be a node of two separate lists (`free_list` and `busy_list`). All list handling functions operate on `list_head_t` structures; you should pass always the pointer to your `list_head_t` member. Once you have a pointer to a `list_head_t` you can get back the pointer to the original object using the `list_entry` macro. See figures A.2 and A.3 for an example.

```
struct data {
  int a;
  int b;
  list_head_t free_list;
  int c;
  list_head_t busy_list;
  int d;
};
```

Figure A.1: Linked list example.

```
list_head_t free_list_head = INIT_LIST_HEAD(free_list_head);

struct data d1,d2,d3;

list_add(&d1.free_list, &free_list_head);
list_add(&d2.free_list, &free_list_head);
list_add(&d3.free_list, &d2.free_list);

// gives: d2->d3->d1

list_add(&d1.busy_list, &busy_list_head);
list_add_tail(&d2.busy_list, &busy_list_head);
list_add_tail(&d3.busy_list, &busy_list_head);

// gives: d1->d2->d3
```

Figure A.2: Linked list ordering.

```
list_head_t *pos;
struct data *current;
int sum=0;

list_for_each(pos, &free_list_head) {
  current = list_entry(pos, struct data, free_list);
  sum += current->a;
}

list_for_each(pos, &busy_list_head) {
  current = list_entry(pos, struct data, busy_list);
  sum += current->b;
}
```

Figure A.3: List traversal.

Figure A.2 demonstrates how a specific ordering can be obtained using `list_add` and `list_add_tail`. Figure A.3 demonstrates list traversal helper macros `list_for_each` and `list_entry`. In order to obtain the original structure from a `list_head_t` you can use `list_entry` macro, passing the node pointer, the type of the structure and the name of the `list_head_t` member.

A.2.2 AVL tree

AVL trees are balanced binary search trees which have $O(\log_2 n)$ complexity. The AVL tree concepts in Acaros is somewhat similar to those of linked lists: the node structure (containing left, right, and skew) is embedded inside the custom data structure. This way a structure can be maintained in multiple trees at the same moment and can put in the tree without allocating node structures (useful when the memory manager is not yet functional).

However, the AVL tree must maintan order and compare elements in order to achieve that. The user must provide a `compare` function and a `match` function.

compare The `compare` function accepts two nodes and returns a value lesser than, equal or greater than zero if the first argument is respectively lesser than, equal or greater than the second argument.

This function is called when a new node is inserted in the tree.

match The `compare` function could be used also for finding an element but the user should create a dummy object with the specified key. However tha main reason to implement a separate `match` function is to implement *range searching* [1].

In order to simplify the syntax of the `avl_add` and `avl_find` macros the `compare` and `match` functions must be named `typename_avlcompare` and `typename_avlmatch` respectively. This naming implies that the type name should not contain spaces[2].

[1] Used expecially in memory management.

[2] structs must be typedefed.Acaros coding conventions already guarantee this.

Appendix B

Coding Style

B.1 Module names

Module names are lowercase.

Example: `hat`

B.2 Function names

Function names are prefixed with the module name and are in "smalltalk-style": every word begins with a capital letter except the first one.

Example: `module_getSomething()`

B.3 Type names

Type names can be prefixed with the module name if the type is private. If types are used by clients but opaque module prefix is optional.

Every type must end with the suffix `_t`. Structure names (not typedefs) may not end with `_t`. Every structure declaration should have a corresponding typedef.