The VAMNET Book

The Virtual Amoeba Machine Environment, AMUNIX and the VX-Amoeba System.

Distributed Programming, Measuring and Controlling Services

Perlimenary Version 1.0.23

BSS LAB

Dr. Stefan Bosse

Table of Content

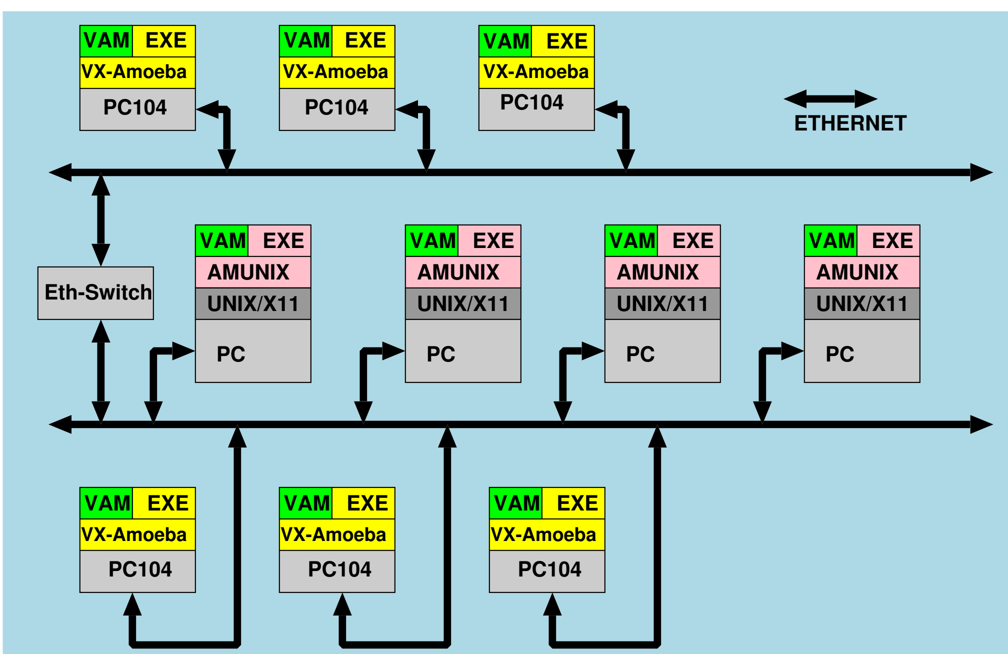
Fundamentals

Overview

The VAMNET is a hybrid operating system environment for distributed applications in a heterogeneous environment, concerning both the hardware architectures used and operating systems already present, for example the UNIX-OS. The VAMNET consists of several parts. Some of them can operate standalone. All of them built up a hybrid distributed operating system environment with some new features never seen before. These parts are:

1. The **VX-Amoeba kernel**, a compact and powerfull microkernel with distributed operating systems features.
2. The **VX-Amoeba environment**, primary consisting of libraries supporting process execution on the top of the VX-Kernel, building a network distributed operating system.
3. The **AMUNIX** environment: Amoeba (concepts) on the top of UNIX like operating systems!
4. The **AMCROSS** crosscompiling environment necessary for building native VX-Amoeba target binaries programmed in C.
5. **VAM**: The Virtual Amoeba Machine. This machine unites the core Amoeba concepts with the world of functional programming in ML and bytecode execution machines for portable and some kind of safe execution of programs. All Amoeba system servers, needed to build up a distributed operating system, were reimplemented with VAM-ML. VAM programs can be executed both on the top of the AMUNIX and the VX-Kernel process layer.

Figure 1 gives a graphical overview of all these components.



(Fig. 1) All the components of the VAMNET system together in an example configuration.

The VAMNET is an ongoing research and development project by **Dr. Stefan Bosse** from the

BSSLAB laboratory, Bremen Germany, started in the year 1999, currently converging to his final stage.

Fields of application

1. Distributed measuring and data acquisition systems, for example remote digital camera servers connected with an ethernet network equipped with digital imaging software.
2. The native Amoeba kernel is very well suited for embedded systems, like PC104 single board equipmment.
3. Distributed systems for machine control.
4. High performance parallel computing and other distributed numerical computations.
5. Distributed filesystems on the top of standard operating systems.
6. Distributed remote (wireless) robot control.
7. Educational tool for the convinient study of distributed services and operating systems.

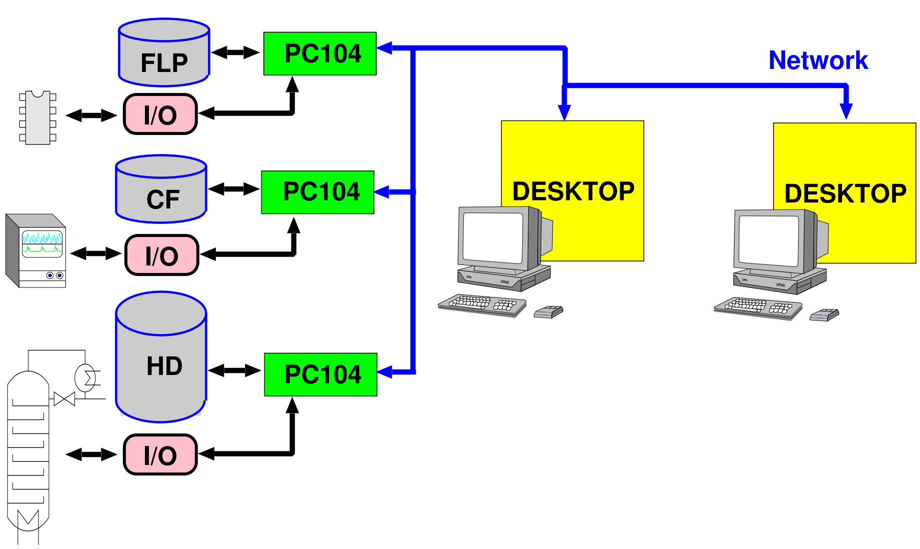
Advantages of a hybrid system

1. The basic concepts of the distrubted operating system Amoeba are avialable with common operating systems with a convinient desktop environment. New operating systems mostly lack of actual device drivers, especially on the i86-pc platform with a wide spectrum of available hadrware.
2. For specializied (perhaps embedded) machines, for example data acquisition systems, or hadware device reduced numeric cluster machines, the native Amoeba kernel is the best choice, featuring a modern and clean microkernel, and exploring the power of the Amoeba system.
3. Both worlds, embedded and specialized computers and desktop computers, can be merged with simple but powerfull methods and concepts using a hybrid system solution. Each machine gets the system which fits best.

The design goals and motivations for such a hybrid system are:

- ➤ A simple method to connect hardware and operation reduced embedded systems, for example PC104 boards, with an already existing pool of computers, for example PCs.
- ➤ On the desktop computer side, the existing operating system must be kept full operable and useable.
- ➤ On the embedded system side only an operating system kernel should be booted (as a starting point) from an arbitrary data storage medium, and not a full sized operating system.
- ➤ The system should be used without the necessity of explicit network configuration. There is no knowledge of the network topology needed.
- ➤ Complete and easy control about the embedded systems by the desktop machines.
- ➤ But all nodes of the system - independent from their hardware resources and type (embedded or desktop) - should be treated as machines with same rights and access methods without handling them within a master-slave hierarchy.
- ➤ The system and processor architecture of the machines may differ. The operating system must be therefore capable of supporting different architectures.
- ➤ Furthermore, it must be possible to execute user programs on different system and processor architectures without recompiling programs for the target architecture.
- ➤ An easy way to implement hardware device drivers without the necessity of understanding complex methods and device driver models inside the operating system kernel and the possibility of rapid prototyping.

Figure 2 shows a common configuration situation of such a system.



(Fig. 2) A common configuration situation. [PC104: embedded system board, IO: external hardware input & output interface, FLP: Floppy drive, CF: compact flash card, HD: hard disk storage]

The hardware reduced embedded systems can be used to control directly special electronics like measuring devices, all kind of laboratory equipment, microcontroller programming, or industrial machines used for example for CNC milling.

The first expected standard solution for network coupled systems is using a UNIX like operating system like Linux or FreeBSD, from now called UNIX for simplicity, on all embedded and desktop nodes. There are many disadvantages of this approach:

- A monolithic UNIX kernel is not standalone operable. It needs a lot of programs running outside the kernel to enable network access.
- This UNIX kernel needs a root filesystem and therefore an integrated hard- or flashdisk or access to a remote filesystem, for example NTFS.
- UNIX is commonly using the TCP/IP network protocol family to communicate between network nodes. But IP needs explicit network knowledge and user configuration. At least something like a Boot DHCP-Server is needed. The IP protocol familiy is not suitable for high performance communication, concerning transfer throuput and latency.
- There is no direct and complete control about an embedded system operating with UNIX from another desktop machine. Some kind of terminal session is nedded like a SSH connection.
- A kernel reboot is complicated: first the kernel image must be saved to the (local) filesystem, then the kernel image must be installed for the boot manager, and the final step is to shutdown the local system, releasing terminal connections from which the system was controlled remotely, and finally the new kernel starts - or crashes. In this case, there is a lot work to reinstall an old working kernel image.
- There is no possibility to reboot a machine with a new kernel directly over the network, to make it easier developing kernel code.
- Distributed process execution is hard to realize and needs special extensions (libraries and programs).
- A UNIX system can only execute binaries supporting the target system and processor architecture the kernel is running on.
- Implementing device drivers can be a complicated and time consuming task, especially for beginners. Addtionally, the device driver interface of UNIX is file operation matched, and mostly not very comfortable for non file and char stream based devices.

But there is a solution avoiding the above explained disadvanteges: the distributed operating system *Amoeba*, originally a research project by the Vrije Universiteit in Amsterdam leaded by the well known Prof. Andrew Tanenbaum and many other people developed this system. The roots gone back in the year 1983, and the research project was canceled in the year 1996.
Concepts and advantages of the Amoeba operating system:

- The network communication is based on a specialized **local network protocol** called FLIP. This protocol is optimized for fast and low latency communication. FLIP needs no explicit network configuration and knowledge about the network topology. Instead, FLIP is able to find routes automatically between communication nodes. In contrast to the TCP/IP protocol, FLIP is connectionless.
- The Amoeba kernel based on **micro kernel concepts**, which makes the kernel more flexible and adaptable than a monolithic one. In contrast to UNIX, the kernel needs no root filesystem on startup. Instead, each kernel has it's own RAM based directory system, mainly used for exporting interfaces of the kernel device drivers and system services.

- **Unique object concepts**: Files, directories, processes, hardware interfaces, memory segments and many more are treated like unique objects with standardized access methods.
- All communication in the system is based on the **server-client modell** using either remote procedure call (RPC) or group communication.

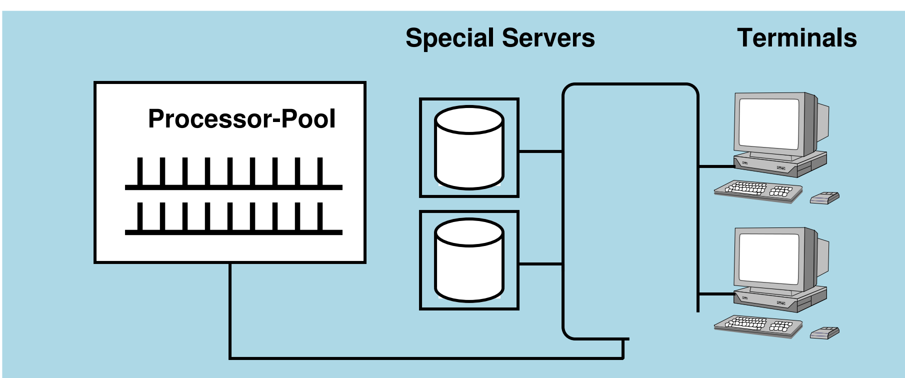
Amoeba forges all machines connected by a network to one distributed virtual machine. Machines of this cluster can serve different tasks:

- File server,
- process server (with or without harddisk storage),
- graphical terminal (X11),
- IO server with spcial devices connected to the machine,
- universal job profile: a desktop workstation (with harddisk storage).

The network topology can be of an arbitrary form. There are no limits concerning the physical and logical media:

- Ethernet: 10/100/1000 MBit/s, star or line architecture,
- Bussystems: VME, Myrinet,...,
- universal interfaces like the serial port, USB, FireWire, CAN-Bus, I^2C and many more.

It's possible to use more than one interface and network on each machine.



(Fig. 3) Machines serving different tasks are connected by a network.

The amoeba operating system based on the client-server model. There are processes providing services, called servers, and there are processes requesting services, called clients. The communication between clients and servers is realized with a unique point-to-point communication using **messages**, called remote procedure call (**RPC**). Beside common point-to-point communications, there is group communication between processes joining a group.
The RPC message transfer is handled with a unique communication header and universal data buffers. Communication with servers mean access of object resources of the contacted server. Amoeba handles all resources with a common resource specifier: the **capability**.
Amoeba objects are:

1. Files managed by the Atomic File Server AFS outside the kernel,
2. directories managed by the Directory and Name Server DNS outside the kernel,
3. processes managed by the process server inside the kernel,
4. data memory space managed by the segment server inside the kernel,
5. hardware devices managed by various servers inside the kernel,
6. terminals (input & output) by the TTY server,
7. UNIX files by the UNIX emulation layer and many more.

In general, objects are an amount of capsulated data. Each object belongs to a server. The capability specifies the server to which the object, for example a file, belongs to, and the access rights associated with this object, for example read and write permissions for files.
Relating to the above shown list of object types, there are different kind of servers handling these objects:

1. The Atomic File Server **AFS** manages files in a very basic way. This server treats all files simply as objects referenced by a number, not a name. There is no name mapping or structureing of files. Additionally, the file data is handled with an atomic behaviour: a file is either valid or not. After a file is marked valid, it must be exists entierly on the permanent data storage and can't be modified anymore! These files can only be read or deleted. Only not valid files can be modified. The name mapping of file objects is handled by the
2. Directory and Name Server **DNS**. It maps object names to capabilities in a gernal way by using a directory like structure method.
   An indepth description can be found in section *DNS: Name mapping of Amoeba Capabilities* (P. 14).
3. A **boot** server to start up, control and shutdown an Amoeba environment. Here, the managed objects are booted programs, for example the file and directory server.
4. Several servers inside the VX-Kernel servicing low level resources.

Servers can be implemented both in kernel and user process space without changes. They use the same RPC communication interface.

Standard Operations

Only certain operations for objects are defined by the server, for example creation of a file or a memory segment. There are some **standard operations** which all server should support:

STD_INFO
Get a short string which holds informations about the accessed object. With this string its possible to identify objects, for example the info string of a directory starts with the '/' char.

STD_STATUS
Get status informations about the server or the accessed object. In general this request returns statistical informations, like the free and used space of a filesystem. The returned string is server specific.

STD_DESTROY
Destroy an already existing object, for exampel a file or a memory segment.

STD_COPY
Make a copy of an object, for example a file, and return the capability of the new created object.

STD_RESTRICT
Request the server to restrict the rights of an object capability, for example making a file readonly. This request returns a modified capability pointing to the same object like the original one.

STD_GETPARAMS
Server operation can be parameterized at runtime. This request returns all supported parameter names and currently values of the server.

STD_SETPARAMS
This request sets server parameters and is needed for system administration of servers at runtime.

STD_TOUCH
Each object has live time value managed by the server to which the object belongs. This request sets the live time to the server internal maximal value. This is one requested needed to implement garbage collection, that means periodical removal of unused server objects.

STD_AGE
Decrease the livetime of all known object of the server by one. All objects with livetime zero will be destroyed by the server upon this call. These are objects which were never touched.

STD_EXIT
Shutdown a server in a clean way.

Remote Procedure Call

The RPC communication under Amoeba needs only three operations to perform a synchronously message transfer from the client to the server with a final returned reply by the server:

1. Server message request operation **getreq(hdr,buf,sz)**
2. Client transaction operation **trans(hdr1,buf1,sz1,hdr2,buf2,sz)**
3. Server message reply operation **putrep(hdr,buf,sz)**

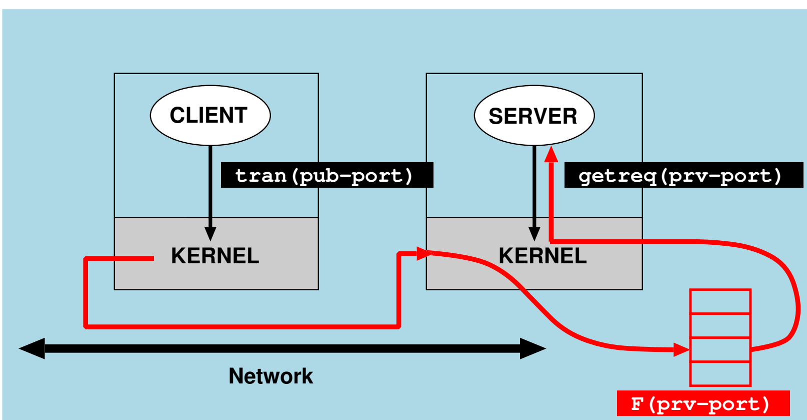
Each communication primitive needs a header which holds informations about the destination of the message using parts from the capability structure. Additonally, in the client-to-server direc-

tion there are informations about the request command and the accessed object, and the reply header holds informations about the status (success) of the requested operation. The message data to be transfered (from client to server and vice versa) is stored in a universal databuffer of specified size. The maximal size of the databuffer is limited to 4 GByte. Because the communication header holds some entries for universal usage, the databuffer can be empty and only the header will be transferred.
The general format of the communication header is shown below.

Communication header

<table>
<tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th></tr>
<tr><td colspan="6">Server Port</td><td colspan="2">Signature</td></tr>
<tr><td colspan="4">Signature</td><td colspan="4">Private Port</td></tr>
<tr><td colspan="6">Private Port</td><td colspan="2">Command</td></tr>
<tr><td colspan="4">User1</td><td colspan="2">User2</td><td colspan="2">User3</td></tr>
</table>

Figure 4 shows a typical sequence of client-server communication. The clients sends a message to the server and gets finally a reply message from the server (not shown).



(Fig. 4) A typcial sequence happening on RPC messaging.

Each server listening for RPC messages has a unique **port address** specifying the server communication endpoint. This port must be public available. To avoid server masquerading, this port address is not published directly. Instead, the server publishes a port address calculated by a one-way function F of his **private port**, called **public port**. Each time a message arrives, the public port in the message header is compared with entries in a server table holding this public port and a pointer to the server waiting for messages of this address. The server always calls the *getreq* function with his private port. The client transaction function uses only the public version. Servers trying to get messages on the public port address will therefore never get a message destinated to the public port.
There is no difference between local and remote communication. In both cases, the RPC primitives can be used. Moreover, the use of this primitives must not know details about the location of a server, in contrast to IP based communication under UNIX.
To make server request more comfortable, there are various so called server stub routines, each performing a special command on a server (object) specified by a capability.

Network protocol FLIP

The RPC messaging and group communication primitives are built upon the Fast Local Intranet Protocol Stack, called **FLIP**. FLIP implements connectionless communication between **communication endpoints** with the following features:

- Communication endpoints are processes, not machines,
- they are identified by 64-Bit IDs, so called Network Servive Access Points (**NSAP**),
- the ID is unique,
- the ID is independent of the location of the process.

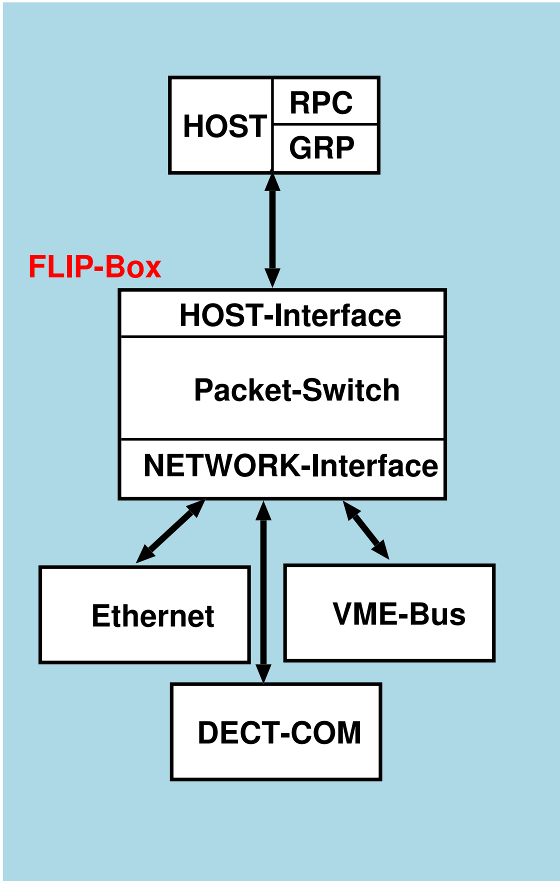
Each Amoeba process keeps his unique communication endpoint identifier, as well the process migrates to another machine!
FLIP has the following **advantages and features**:

- optimized for low latency and high data throughput rates,
- independent of the network topology and the used physical transmission lines,
- and it's simultaneous a network package router,
- FLIP is a reliable and connectionless message protocol,
- both point-to-point and muclticast transfer (group communication) is possible,
- FLIP adapts dynamically and automatically to network topology.

Routing takes place

- between different networks,
- between different physical medias,
- and the best route will be automatically determined (latency $\otimes$ data rate).

Each higher level messaging system (RPC/GROUP) is treated by FLIP as another network. Therefore, a FLIP box always performs routing between networks for delivering messages. Figure 5 explains this relationship.



(Fig. 5) Details of a FLIP box.

Each FLIP box contains a packet switch with a routing table, but known routes are not binding. Is a destination not reachable, the FLIP box tries to locate a destination again, perhaps using other routes. Networks will be weighted by their power and the best connection is choosen for message passing.
More details about the FLIP protocol can be found in section *FLIP* (P. 42) covering results from the original authors.

As mentioned in the previous section, server resources are specfied with capabilities. A capability therefore holds the server port and additional informations about the object. The following graphic shows the structure of a capability:

Object Capability

Port	**Object**	**Rights**	**Private**
P1:P2:P3:P4:P5:P6	obj	<rgts>	S1:S2:S3:S4:S5:S6

The entries have the following meanings:

P
: The public server port (6 bytes),

obj
: the object number specified by the server (4 bytes). It's a unqiue server internal identification number of this object,

<rtgs>
: the rights mask (1 byte) determines the allowed access rights of an object, like the permission to destroy an object. Each bit in the rights field specfies one possible access right. The meaning of each bit is really dependent of the server and the kind of the object.

S
: and finally the security private port (6 bytes). This port protects the rights field against manipulation.

The rights protection port contains the rights field. This is done by a one-way encoding function f using a private check port C randomly created by the server only for this object and the rights field. A restricted capability CAP' is built from an original one by restricting the rights field and creating a new security port using the *prv_encode* function. This function simply calculates the new security port S' from the private checkport C and the rights field R using a logical XOR operation and feeds the result to a one-way function F, as shown in equation 1.

$$S' = F(S \textbf{ XOR } R) \qquad (1)$$

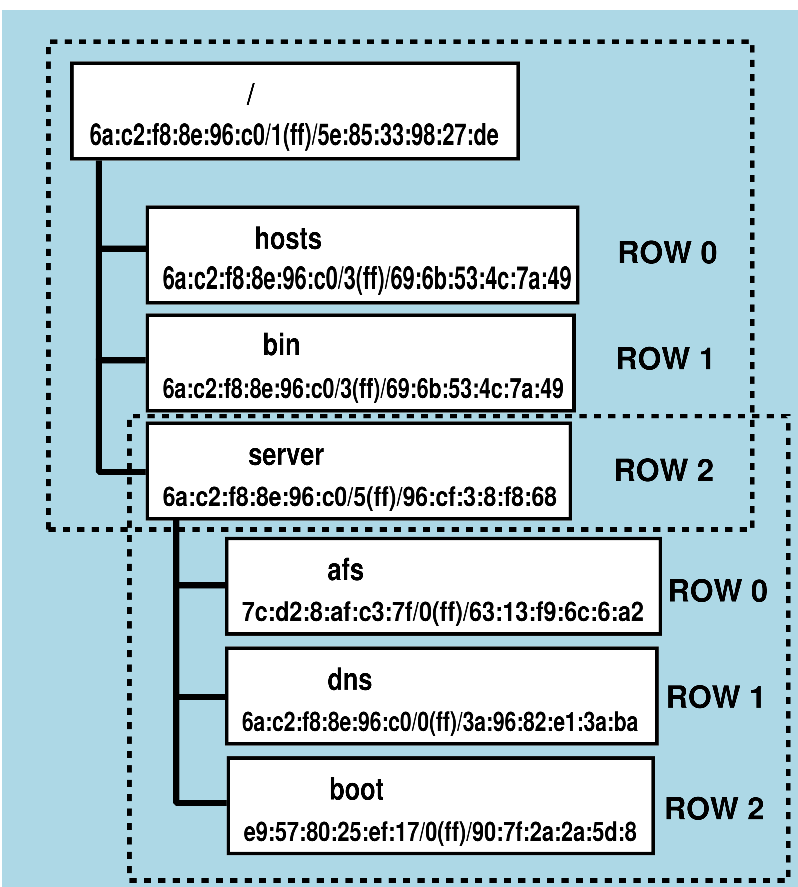
Each time a server receives a message it checks the security port using his private check port C and the *prv_decode* function. This function simply builts the expected security port S' from the rights field specified in the received capability and the checkport C and compares S' with the supplied S port. If they are not equal, the capability was manipulated and will be rejected.
Capabilities can be reperesented in text form in the format shown above (server port, object number, rights, private protection port):

```
6a:c2:f8:8e:96:c0/1(ff)/5e:85:33:98:27:de
```

The previously shown capabilities are always needed for requesting services and handling Amoeba objects. But they are not a human friendly way to handle Objects. Therefore, some kind of a name-to-capability mapping is needed. This is performed by the Directory- and Name Service (DNS):

- ➤ The DNS is build from directories with a definite number of entries, called rows,
- ➤ each row entry maps a user specified name string to a pair of capabilities, but normally only one capability entry is used (the other is used for a replicated version of an object, serviced by another server for redundancy),
- ➤ additionally to the row name mapping, there are so called columns in each row, which determine objects rights,
- ➤ the DNS directory is managed by a dedicated server independent from file servers,
- ➤ each directory is handled standalone, that means, there is no parent directory feature in the DNS,
- ➤ and finally each directory has it's own capability, like any other object in Amoeba.

Figure 6 shows an example DNS structure configuration.



(Fig. 6) An example for a directory structure in Amoeba.

Because each directory in the DNS is handled standalone, each directory can be a new root directory for a user, not only the directory named ”/”.
There are variuous library routines which make teh access of the DNS service comfortable, like the name module:

```
name_lookup("/server/dns")      →
cap     ≡    6a:c2:f8:8e:96:c0/1(ff)/5e:85:33:98:27:de

std_info ( cap     ≡    6a:c2:f8:8e:96:c0/1(ff)/5e:85:33:98:27:de )      →
"DNS server capability"
```

The way object rights are determined (that means the operations and requests allowed with this particular object) is different from UNIX like operating systems. A typical Amoeba directory may look like:

Directory content ”/”

Name	**Capability-Set**	**Col1**	**Col2**	**Col3**
hosts	C1,C2,C3	1111	0001	0000
bin	C1,C2,C3	1111	0001	0000
server	C1,C2,C3	1111	0000	0000

Here, the three rights columns (with only four bits shown) of the DNS server have the symbolic names: ”Owner”, ”Group”, ”Others”. If a directory is accessed, the (first three) bit of the rights field from the supplied user capability determines the access rights to each column of a row. The first bit corresponds th Col1 with capability C1, the second to Col2 with capabilty C2, the third to Col3 and the C3 capability. This is a convention by the DNS server, and not a general method for rights handled by other kinds of server, but more than three rows with different meanings can be implemented using DNS. Each column represents therefore a (rights restricted) capability of the object with the rights specified in the particular column.
Now suppose, a user want to access the entry ”bin” in this directory. In the first case, he owns the unrestricted capability of this directory (all rights bits are set, here ”0xff”):

```
6a:c2:f8:8e:96:c0/1(ff)/5e:85:33:98:27:de
```

First the DNS server will lookup the appropiate row specfied by the requested name. Next, the column is checked against the bit map in the user supplied rights field, to see which column(s) should be used. All the columns rights from the current row are logical ored, if, and only if the i-th bit in the current rights field is set. The i-th bit corresponds to the i-th rights column in the current row:

```
have_rights = 0
for all columns in row
do
    if bit i in rights is set
    then have_rights = have_rights lor col[i]
done
```

With the unrestricted rights field (111) we get a capability with rights (...1111):

```
6a:c2:f8:8e:96:c0/3(ff)/69:6b:53:4c:7a:49
```

Now, the user has only a restricted version of the directory capability:

```
6a:c2:f8:8e:96:c0/1(2)/1e:45:aa:81:16:ae
```

This leads to another calculation of the capability rights (here 0001) and another restricted capability of the directory ”bin”:

```
6a:c2:f8:8e:96:c0/3(1)/1a:23:17:94:45:91
```

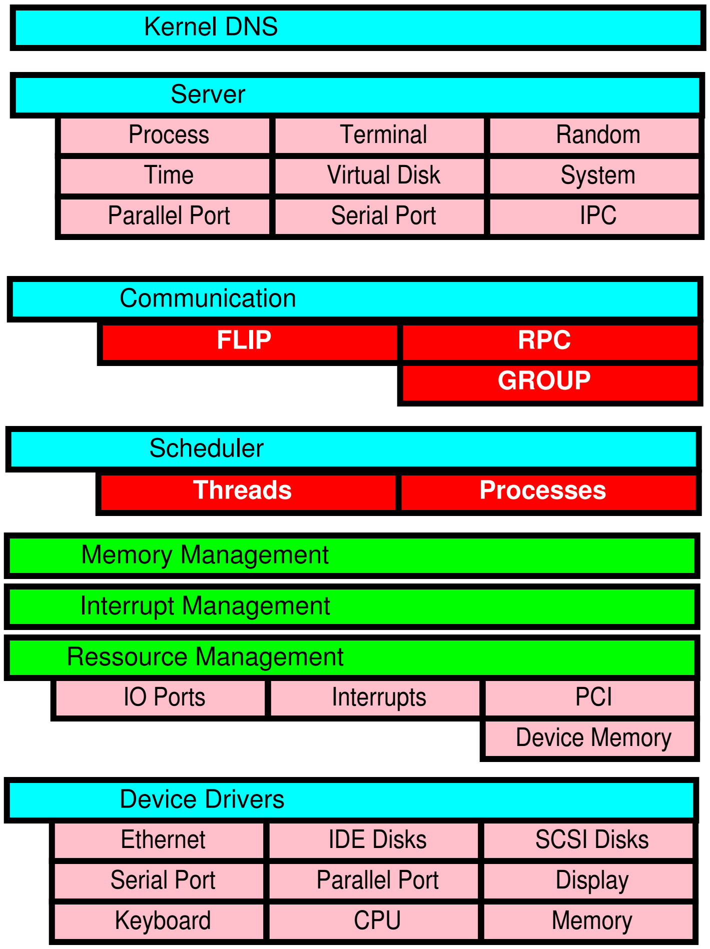
You see that all capabilities have the same server port and object number, but different rights leads to different private security ports which contain the rights coded with the already explained cryptho-graphic scheme.

The VX-Kernel is derived from the original Vrije-Amoeba kernel and it is a native Amoeba execution platform with its own set of device drivers and low level resource management. The VX-Kernel is used by the VAMRAW system, providing virtual machine concepts and functional programming on the top of this kernel.
The kernel has the following features and avtanges:

- it's a micro kernel (with some advantages of a monolithic kernel like a simplified boot operation and core device drivers inside the kernel), and can be extended and scaled to customized designs,
- it supports true multiprocess execution in a protected user process environment and multithreading, both inside the kernel and user proceeses,
- the kernel has full control about low level process and thread management,
- device drivers either built in the kernel or executing outside the kernel as normal protected processes,
- segment based memory management with low level architecture dependent page protection, which protects processes against each other, which protects the kernel against process memory violations (which leads to process abort), and finally, protects processes against kernel memory protection violations (which leads to kernel abort),
- the kernel has a two-level priority based process and thread scheduler, but inside the kernel threads are non preemptive scheduled,
- the thread and rpc programming interface is the same both inside and outside the kernel,
- a restricted version of the Amoeba core library is available inside the kernel,
- the kernel supports different communication facilities: the common RPC interface (for both local and remote message transfers) and a specialized local process interface IPC, similar to RPC, but with enhanced performance,
- to enable high performance local and remote interprocess communciation, the FLIP protocol stack is part of the kernel.

Figure 7 gives an overview of all the components inside the kernel. Most of them are necessary for a fully functional kernel.
Each kernel has its own internal and pure memory based **directory and name service DNS**. Only a small part of system access takes places using the kernel system call interface, a direct path to the kernel simply calling a system function. In contrast to monolithic operating systems like UNIX most of the system access like rading files is implemeneted with message passing. The remaining system calls of the VX-Kernel are used for:

- low level thread and process management,
- low level memory management,
- user space device driver support (like user process interrupts),
- and finally the few functions (getreq,putrep,trans) of the RPC message interface (additionally the group communication interface).



(Fig. 7) Overview of the VX-Kernel structure.

The **servers inside the kernel** managing the system resources are requested with remote procedure calls, like all other servers in the Amoeba system running in the user process context. But, its not necessary to have a local filesystem supported by the kernel. Therefore, all kernel servers publish their public server capability in the kernel DNS. Most server generate their server port randomly on kernel startup, except disk servers. They save their server port on the disk they are using, but only if an Amoeba fielsystem is located on the disk.
The different servers inside the kernel are:

1. The **system server (sys)** provides generic system informations (kernel statistics) and system services like the reboot feature,
2. the **virtual disk server (vdisk)** providing a unique access to storage devices, used by higher level filesystem servers like AFS,
3. **time and random number services (tod,random)**,
4. a low level **process server** (proc), which publishes the process capabilities in a special directory called ”ps”,
5. and many **device drivers** like the terminal server (keyborad, display: TTY), parallel and serial port and many more.

Below the server and device driver layer there is the heart of kernel loacted: the hard- and software resource management (slightly distributed over several parts of the kernel). The following main resources are handled:

IO
: Hadrware IO ports of machine devices

IRQ
: Interrupts of hardware devices. The same interrupt level may me shared between several devices.

VMEM
: Virtual memory. Each process (inclusive the kernel) has its own virtual address space. But in contrast to most monolithic systems there is no swapping of virtual memory parts to a secondary storage media. This limitation results from first the overhead needed, second the fact, that RAM memory is today available in amounts sufficient for most applications, and finally third the communication system performance decreases with outsourced memory parts considerable.

PMEM
: Physical memory management with access protection features.

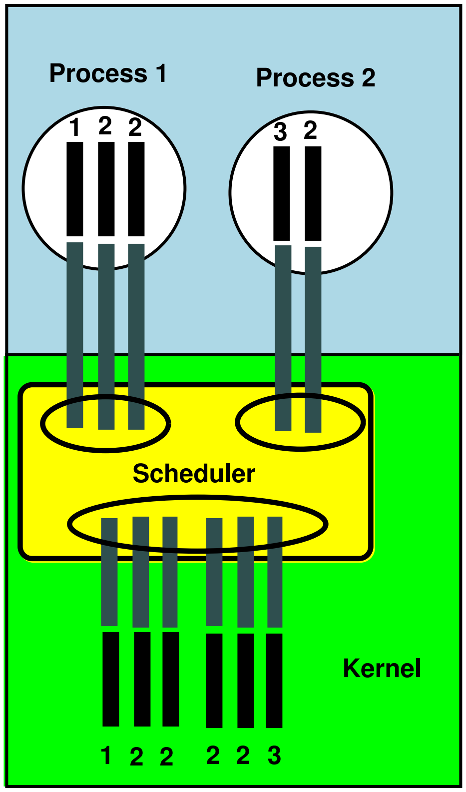
PCI
: Special bus systems resources like the PCI bus (memory, interrupts, IO ports) need configuration and management.

High level Interrupt management is different for device drivers inside and outside the kernel. Within the kernel, the device drivers simply provides a function and installs it as an interrupt handler. If the hardware interrupt was triggered, the kernel will call the device driver interrupt handler function directly. But the process context of such an interrupt handler depends on the current executing process! The interrupt handler function can for example overflow the stack of the current process (thread).
Outside the kernel, in user processes, there is another solution. The device driver starts a thread servicing the desired interrupt. This thread must register the interrupt and waits for the interrupt event calling a special interrupt await function blocking the thread untill the interrupt occurs. If the interrupt was triggered, the kernel will wakeup this thread, and the device driver can service the interrupt. These interrupt handlers always execute in their own process context, which make the interrupt service much more safety.
Another important part of the kernel is the **time(r) management**. In contrast to traditional kernels has the VX-Kernel a dynamically timer management, that means there is no fixed time unit (ticks). The two jobs of the kernel timer management are:

1. periodically call user supllied timer functions,
2. and handling of thread and process timeouts.

The internal (theoretical) time resolution is about one microsecond. The shortest time intervall needed is determined dynamically on demand. The timer management is hardware interrupt controlled. After a programmable hardware timer triggers an interrupt because the programmed time interval T expired, the timer manager *timer_run* will be called and thread, process and user function timeouts $T_i = T_i - T$ will be calculated. Functions ready to run (timeout $T_i < 0$ reached) will be executed. Finally, a new timer intervall T (of the timer manager itself) will be calculated and programmed into the hardware timer. The timer manager is weaved with the scheduler (for thread and process timeouts).
The **thread management module** in the VX-Kernel was fully revised and differs interally from the

orginal Amoeba kernel, but the programming interface kept nearly unchanged, except some enhancements for therad creation.
A process consists initially of one thread, the main thread. Each process, the kernel is terated like a process, can start new threads. Each threads has its own stack. Figure ?? shows a typical situation.



(Fig. 8) Thread and processes in the VX-Kernel with different priorities.

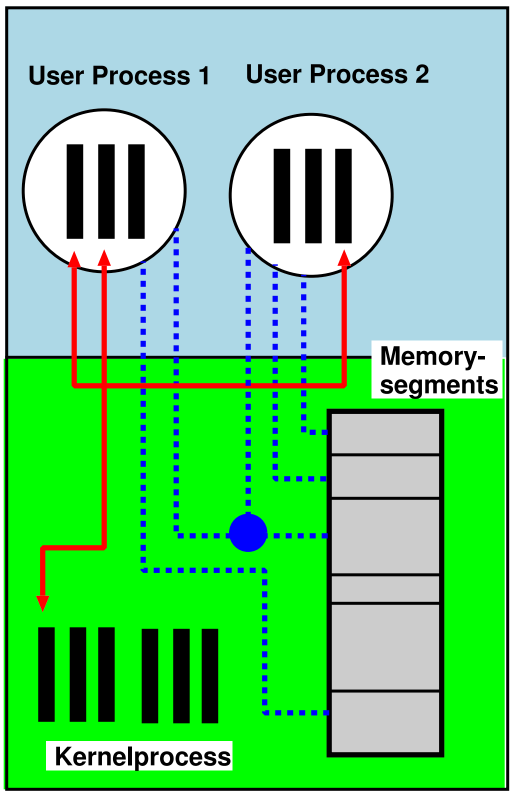
The thread and process **scheduling** is based on a two-level proirity scheme:

1. Each thread of a process has a thread priority TPRIO which can have the three different values:
   TPRIO=$\{$**HIGH,NORM,LOW**$\}$
   The thread priority has a process local context, that means that each process can choose his thread priorities without limitations.
2. Each process has a process prioritry PPRIO which can have thre different values, too:
   PPRIO=$\{$**HIGH,NORM,LOW**$\}$
   The process priority has a higher weight than the thread priority.

Kernel threads are scheduled strictly non preemptive, but priority selected. User process threads can be scheduled either preemptive or non preemptive (the default setting). A kernel thread run as long as it calls a function which blocks the execution of the thread (trans, await, thread_switch...). User processes are scheduled preemptive with a time slice. If a process has consumed his time slice, another process (by priority) is selected. The kernel process has the highest priority HIGH.
The memory of a process (and the kernel) is structured by segments. A process has at least three memory segments:

1. The textsegment (readonly RO) which holds the program code and constant data of a program like strings,
2. at least on data segment (with read&write rights RW),
3. and at least one stack segment (RW).

All the memory segments are handled by the segment manager as part of the system server. Each process can allocate more memory segments, for example using the *malloc* function, which request a new data segment from the segment server (via a system call). Each new created thread gets his own stack segment. Figure 9 shows the usage of memory segments. Memory segment can be shared between processes executing on the same machine. One common example is the text segment of a process.



(Fig. 9) Memory segments used by processes and the kernel.

To synchronize the runtime behaviour of different threads inside of one process, there are different mechanisms:

- With **Mutual Exclusions** (Mutex) shared data structures can be protected against uncontrolled shared access,
- with **semaphores** a producer-consumer algorithm can be implemented,
- **barriers** can synchronize the execution of several threads,
- **signals** can be used to communicate between different threads of a process,
- and an **await-wakeup** implementation is used for the simplest interthread synchronization.

Process synchronization takes place with:

- Remote Procedure Calls **RPC**, both for the local and remote case,
- and local interprocess communication **IPC**, normally only used by devicde drivers outside the kernel.

Each Amoeba process is handled with a so called **process descriptor**. This is data structure which contains the following informations:

1. The processor and machine architecture for which this process binary was compiled,
2. an owner capability,
3. a list of memory segments (at least the text segment),
4. a list of threads currently executing with additional informations about the thread state.

The **segment descriptor** holds these informations:

1. The segment capability (specifying the owner of the segment),
2. the start address and size,
3. status flags of a segment (RO/RW/SHARED...).

Finally the **thread descriptor** holds informations about the current state of each thread of a process:

1. The program counter IP,
2. the stack pointer SP,
3. processor register copy,
4. flags and signals,
5. and finally architecture dependent data, for example a fault frame after an exception was raised.

The process descriptor is part of each program binary file with informations about the text, initialized and uninitialized datasegments, and the main stack segment. The owner of these segments stored in the binary is the fileserver untill the process was started.
A process is started by another process (or the kernel for booting) simply by calling a process execution function with the process descriptor read from the binary file. The process creator will be the owner of the new started process. The stack segment, which need to be initialized with the process environment, like environment capabilities and strings, is created using Amoebas fileserver, simply by creating a new temporaryly file. This must be done by the process creator. So, the low level process server reads all segments content for the new process from the fileserver, just by examining the segment descriptors and extracting the owner capabilities.
A running process can be dumped together with his process descriptor to an imagefile and be restarted on another machine simply calling the process execution function again with the current process descriptor.

The **process environment**, committed to a new process by his stack segment, contains the following informations:

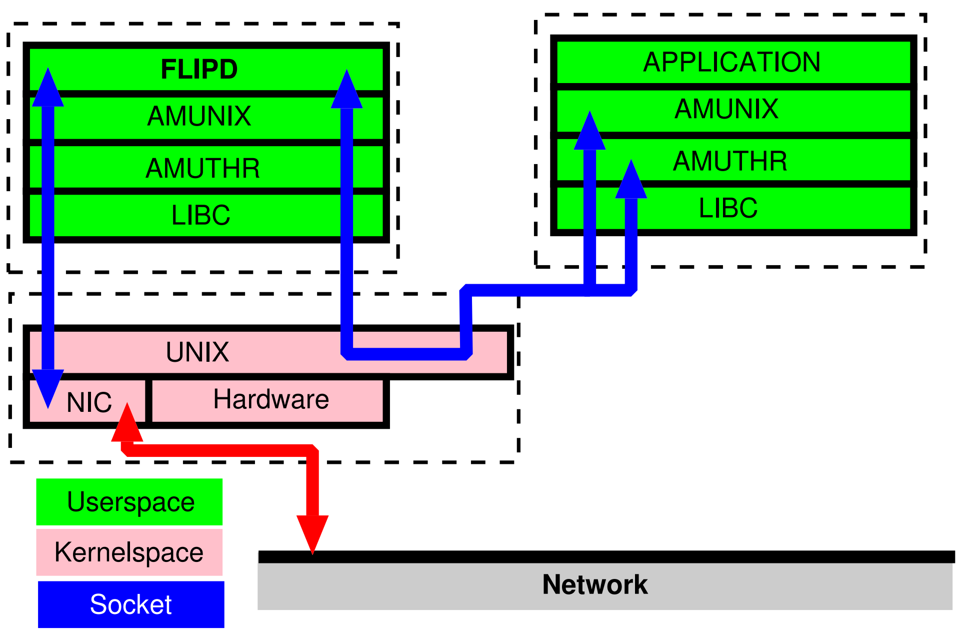
1. program arguments supplied by the user,
2. standard capabilities:
   - TTY: terminal server for standard output and input,
   - RANDOM: random generator server,
   - TOD: time server,
   - ROOT: the capability of the root directory for accessing files and directories.
3. string variables, like the terminal type TERM.

The FLIP protocol stack was fully revised and split in an operating system dependent and an independent part. Most of the source code is now fully operating system independent and is shared in the kernel and the AMUNIX implementation.

The AMUNIX system provides the interface to the basic Amoeba concepts like RPC messaging on the top of UNIX like operating systems, for example the open source Linux and FreeBSD operating systems. It consists of these parts:

1. An UNIX version of the Amoeba thread module called **AMUTHR** enabling multithreading normally not a core part of a UNIX operating system. The AMUTHR module is entirely implemented in UNIX user process space and nearly 100% compatible to the Amoeba kernel thread implementation. The Amoeba thread implementation is weaved with the FLIP protocol stack.
2. The **AMUNIX library** implementing Amoebas basic concepts like capability management or the RPC interface, several server stib functions and many more. It's mostly derived from the native Amoeba core library.
3. The **FLIPD protocol stack** daemon entirely executing in the UNIX user process domain providing access of AMUNIX programs to the (Amoeba) network using the FLIP protocol. The FLIPD is also responsible for loacl communication between Amoeba programs.
4. A comfortable **development environment** with predefined makefiles for the Amoeba configuration manager amake similar to UNIX make. With this development environement it's possible to built libraries and Amoeba executables from C source code.

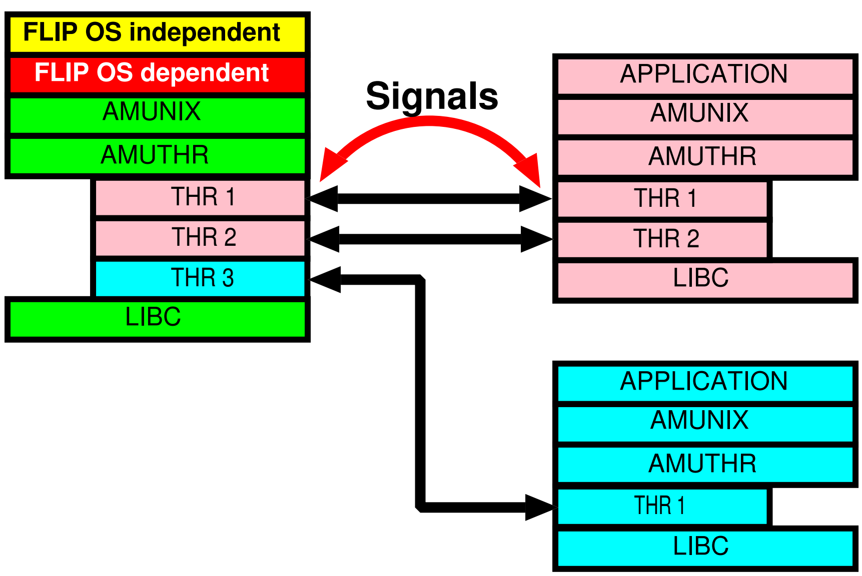
Figure 10 gives an overview and the relationship between these parts.



(Fig. 10) AMUNIX: the Amoeba layer on the top of UNIX.

Each AMUNIX executable incorporates the underlying UNIX system, glued in the system C library. On the top of this system library, the AMUTHR module was placed to enable multithreading. **Each AMUNIX process has it's own encapsulated thread management responsible only for this process.** This mechanism differs from the native Amoeba system (VX-Kernel) where all processes share the same thread manger inside the kernel. The interface to the Amoeba world is provided by the

AMUNIX layer.
The AMUTHR thread module is nearly identical to theVX- kernel thread implementation. Yes, indead, the source code from the kernel were used nearly unchanged. A thread switch is performed by a small function written in Assembler, consisting of less than 10 lines of code. Only the stack and program code pointers must be changed during a thread switch.
The AMUNIX library differs from the native Amoeba core library only in the thread and the communication backend. Under native Amoeba with the protocol stack inside the kernel, the communication backend (RPC...) is implemented simply with kernel system calls. Under AMUNIX, a UNIX like communication must be established to the FLIPD daemon, an AMUNIX process, too. The communication between AMUNIX processes and the FLIP daemon is realized with generic UNIX sockets.
Using the AMUNIX layer, nearly all Amoeba programs known from the native System can be build for the AMUNIX environment. The programming interface kept unchanged including C header files. Only different libraries must be linked with the AMUNIX executable. And finally, an AMUNIX program can be started from any UNIX shell or forked by another UNIX program.
The first time an AMUNIX thread want to use the RPC interface (e.g. with a *trans()* call), the AMUNIX layer will try to connect to the FLIP daemon via a public UNIX control socket. After successfull connect, the FLIP daemon will establish a new private communication socket only for this particular thread. Additionally for RPC signal transmission, a dedicated signal socket connection is opened if this is the first thread of the process. Each AMUNIX process thread (using RPC) is handled by the FLIPD daemon with an own process thread.
Each AMUNIX process (like a native Amoeba process) must be registered by the FLIP protocol stack. Each Amoeba process gets an unique communication endpoint identifier. As long as the process lives, it keeps this ID number, as well the process migrates to another machine!
Figure 11 shows details of the FLIP daemon operation with AMUNIX processes.



(Fig. 11) The UNIX implementation of Amoebas FLIP protocol stack in the UNIX user process space.

The **network interface** of the FLIP daemon needs direct access to the network layer of the underlying operating system to receive and send raw ethernet packets. Therefore FLIPD is connected to the UNIX kernel using some kind of sockets, too. This is the only host system dependent part of the FLIP daemon program. Under Linux, so called raw sockets, and under FreeBSD the packet filter interface is used to connect FLIP to the network. The rest of the FLIP source code (both for the native VX-Kernel and AMUNIX implementation) is operating system independent.

The short name VAM is the abbreviation for the **Virtual Amoeba Machine**. The concepts of a native Amoeba system running with its own kernel and the AMUNIX Amoeba emulation as an addon for UNIX operating systems have many advantages. But there are some important disadvantages of these traditional concepts:

1. All parts of the kernel, the library and user space programs are programmed in the C language. C is a powerfull lowlevel language, but it lacks of safety and the ability of abstraction. There are too many risks of failures, mostly related with the always visible pointer handling. Moreover, the programmers spent a lot of time and power with resource management, like memory space for data structures.
2. C programs are compiled and linked for the native microprocessor code only. This is no problem for the (of course portable) VX-Kernel, currently only supporting the i386 architecture. But the AMUNIX emulation should run on various different operating systems and microprocessors. For n different operating systems and m hardware architectures, youn need n$*$m builds of the AMUNIX system, all the libraries and util programs, the flip protocol stack and so on.

Some disadvantages can be eliminated with new and modern concepts of **functional programming** instead using memory pointer based languages like C with a high probability to cause unresolved errors in the program code during program execution somewhere in time and space. The authors experiences showed in the past, that programming errors due to wrong pointer handling can occur years (!!!) after the program was written. These class of programming errors are really hard to find out, because wrong pointer handling causes normally no exception directly. Instead, some part of the program memory will be corrupted. This will cause a fully undefined program behaviour with program aborts in parts of the program not related to the original pointer corruption. More than 90% of the C code in the world is infected by pointer corruption and executes in an undefined way.
Functional programming, especial the **ML programming language**, features a strong typed data system which avoids several common programming errors, like integers of different binary widths which can cause unpredictable program output. Moreover, ML provides the programmer with an automatic resource management. There is no memory pointer code needed (and of course allowed) to program an algorithm. True functional code has a predictable runtime behaviour.
In contrast to the machine C language the ML languages provides the programmer with some kind of abstraction facilities. Functional programming is a style that uses the defintion and application of functions as essential concepts *[COU98]*. In this approach, variables play the same role as in mathematics: they are the symbolic representation of values and, in particular, serve as the parameters of functions.
In imperative programming languages like C there are some parasites: each function must be introduced with an arbitrary name F. In those languages, we cannot define a function without naming it and there is an explicit assignment operation, like the return operation. Using imperative languages, the user must provide the compiler with the type of functions or variables. In contrast, in functional programming the compiler is responsible to find out types of functions and variables (or data structures)! **Types exist in ML, too, though they are computed by the system.**
A functional style tends to preserve the simplicity and spirit of mathematical notation. Because functions can be denoted by expressions, function values can be treated as values just like all others and therefore functional values can be passed as argumentes to and returned by other functions.
For reasons explained below, the **OCaML programming language** from INRIA software institute was choosen for implementing the VAM system. OCaML is a kind of ML language, but not fully compatible to standard ML. It provides a ML core with a powerfull and easy to use module system. Additionally, it has an object orientated class system built on the top of the ML core.
To illustrate the power of functional programming, lets make an example:

```
fun x → 2 * x + 1
```

This is simply an expression for a mathematical function. Using C, we need to write instead:

```
int F(int x){
   return (2*x+1);
}
```

A named function value, for example ”f”, can be expressed with the *let* operator:

```
let f x = 2 * x + 1
```

The ML compiler will compile this expression and a result is the following derived type interface:

```
# val f : int → int
```

You can see, that there is no necessatiy of a type declaration if you define a (functional) value. The compiler will evaluate the expression and finds out that the multiplication and addition operator is from type integer (int), and therefore the function argument and the result of the function must be from type integer, too.
The fact that the ML language treats functions as generic functional values like normal ”variable” values can clearly shown by the next two examples:

```
let f x = x + 1
# val f : int → int
let x = 2
# val x : int
```

Here, the first line defines a function expecting one argument, and the second definition just defines a symbolic variable (not mutable) from type integer. This value just returns a constant.
Another powerfull of ML is **polymorphism**. That means, the compiler can't evaluate one or more types of functional values inclusive the return value of a fucntion. This leads to a powerfull and mighty instrument for reuseable code. For example a function which iterates a list entry by entry and passes each list entry to a user supplied function which make some nice things with this list entry:

```
let rec list_iter func list =
   match list with
   | hd::tl → func hd; list_iter func tl;
   | [] → ();
;;

# val list_iter : ('a → 'b) → 'a list → unit = <fun>
```

The interface derived by the compiler specifies no concrete type. The only fact we know is that the first argument must be a function wich expects as the first argument a list entry from same type as specified in the second list argument `'a list`. The native lists supported by ML are simple single linked linear lists. Lists can hold data of arbitrary types. The :: operator in the match statemant, compareable with the switch/case statemant in C, splits the list into a single value, the head of the list, and a remaining tail list. The [] operator is the empty list. The last example showed one more feature of the ML language: **function recursion**.
Another kind of data type leads to a more strcutured programming style: tuples. These are simply spoken unnamed compounds of arbitrary data types and entries. Lets assume you want to return more than one value from a function. In C you must use several pointers passed to the function with its arguments. But clearly, function arguments should of type input, not of type output. Using ML, there is a solution, called data tuples. The following example shows this powerfull feature, with a function expecting to arguments of type integer, and returns a tuple of thre integers:

```
let arithm x y =
   let mul = x * y in
   let div = x / y in
   let add = x + y in
   (mul,div,add)
;;
# val arithm : int → int → int * int * int = <fun>
```

A final example shows the usage of the above defined polymorph function *list_iter*:

```
let list = [1;2;3] ;;
# val list : int list = [1; 2; 3]

let sum = ref 0 ;;
# val sum : int ref = {contents = 0}

list_iter (fun x → sum := !sum + x) list ;;
print_int !sum
# 6
```

The first line defines a list with three entries from type integer. The second one defines a traditional variable known from imperative programming languages. This imperative variable is mutable, as shown in the nameless function in the *list_iter* function evaluation call. The "!" operator just returns the current value of this variable, and the ":=" operator assign a new value to the variable. But in fact this is not a traditional variable, it's a reference to an object. Each time a new value is assigned to this reference, this reference points to a new object! In the above example it's just the result of the addition of two values - a constant in this case.
But back to the motivations for using functional programming for the extensive job of operating system programming. As shown above in various examples, the **programmer spent his time with implementing an algorithm**, and not with resource allocation and release, like in imperative languages like C. This make especial rapid prototyping more faster and safe. This can lead **to a more clean and structured programming** style. In CaML there are references, too. But they must point always to valid objects. There is no nil pointer like in C. And therefore memory access violations due to nil pointers are not possible. This reduces the time spent in the job of programming of about thousand of hours, really belive it.
But that's not all, folks. The OCaML language has one more powerfull feature: the ML compiler

doesn't produce native assembler code directly executed by the host machine, no, it produces architecture and machine independent code, called bytecode. This bytecode is then interpreted by a **virtual machine** , emulating an abstract and in the case of OCaML nearly perfect and to the ML language highly adpated execution machine. This virtual machine hides all system dependencies from the underlying host operating system. This feature is perfectly suited for the implementation of a **portable operating system environment** !
The OCaML virtual machine is a traditional stack based bytecode processor with memory allocation and delayed freeing of no more needed memory by a background garbage collector. Experiencences showd that an algorithm executing with OCaML using the virtual machine approach is only about 4-5 times slower in execution time than an optimized C program. The required memory space is hard to predict, perhaps in contrast to C programs. It can be of course higher than by a compareable C program. So, for embedded microcontrollers with hard resource constraints, the C language is mostly the better choice.
Now, with the functional approach in mind, it seems to be a simple task to implement distributed operating system concepts using the OCaML VM and the ML language. Indeed, the original OCaML virtual machine is highly portable. The virtual machine consists of these main parts:

1. The bytecode interpreter with a stack based CISC machine architecture. It's mainly one C function unrolling all the bytecode instructions (about 140 instructions),
2. several hardcoded ML standard function groups: Arrays, Lists, Strings, Integers, Floats...,
3. support for custom datastructures not interpreted by the VM (handled in external fucntions),
4. the memory manager and the garbage collector,
5. some system dependent parts (the Unix and System module),
6. IO handling (terminal and file input & output),
7. backtracing and debugger support.

One result of the functional approach of ML is that functional values can be evaluated independently. This offers a great advantage for interactive toplevel systems. OCaML is equipped with an interactive interpreter system. You can either type instructions on the input line, or read input from a file. This text input is compiled (evaluated) to bytecode and can be immediately executed. These on the fly compiled code executes with the same runtime behaviour than traditional bytecode externally compiled directly using the compiler.
The OCaML system, both the virtual machine and the runtime system (VM) , was adapted to the demands of the Amoeba operating system concepts. The VM was improved, and the bytecode compiler gots some enhancements.
One main feature of the OCaML virtual machine is a simple interface of user customized C functions accessible from ML code. For example a function allocating a new Amoeba port is implemented in an external C function:

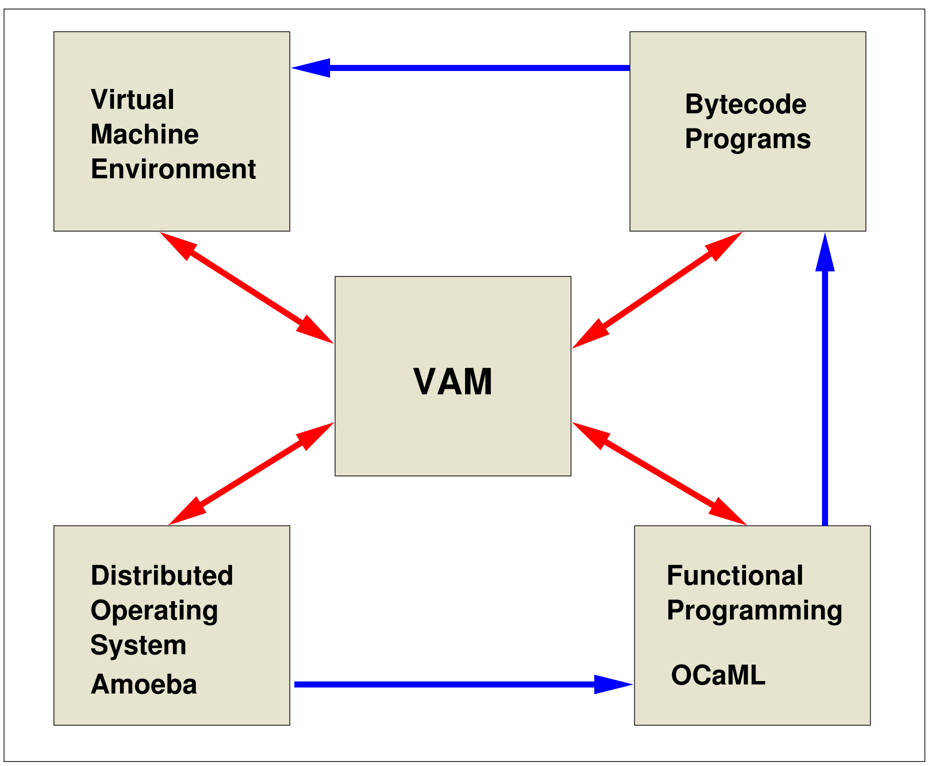
```
CAMLexport value ext_port_new (value unit)
{
    CAMLparam0();
    CAMLlocal1(port_v);
    CAMLlocal1(port_s);
    port_v = alloc_tuple(1);
    port_s = alloc_string(PORTSIZE);
    memset(String_val(port_s),0,PORTSIZE);
    Store_field(port_v,0,port_s);
```

```
    CAMLreturn(port_v);
}
```

The ML code, which tells the compiler that the function is external, is now very simple:

```
type port = Port of string (* length = PORTSIZE *)
external port_new:  unit → port = "ext_port_new"
```

Each time the *port_new* ML function is called, the virtual machine will call the *ext_port_new* C function. The virtual machine is implemented only with a library and a main source code file created dynamically. Therefore, the virtual machine can be recompiled any time with additional functionality. This feature was an important starting point for VAM.
Most of the Amoeba modules known from the C world were reimplemented with ML. Only a small part is linked inside the virtual machine, namely the AMUNIX interface for threads and RPC communication. All other parts are created from pure ML source.
The basic concepts of the VAM system are shown in figure12.



(Fig. 12) The basic conecpts of the virtual amoeba machine.

The **VAM system** is divided into a development and a runtime environment. The development envrionmnet provides the following parts:

1. Several **ML libraries** with different modules implementing the **ML core** concepts, the interface to the **UNIX** environment, the **Amoeba** system, building the largest part, and a **graphical widget** library.

2. A standalone **ML Bytecode compiler** producing bytecode exectubales. Additionally, this compiler can compile a new custom designed virtual machine. The bytecode compiler is completely programmed in ML, too.
3. An **interactive toplevel VAM** program. This program, simply called *vam*, contains the bytecode compiler, a command line like toplevel shell for user interaction, and all ML Modules. With this interactive system it's possible to compile expressions directly entered into the command line or load external ML scripts, compiled on the fly, too. This on the fly compiled bytecode is linked to the current bytecode program during runtime and can be executed like any other built in function.

The following list (in alphabetic order) gives an overview of currently implemented VAM modules. Some modules were provided by external programmers. They were modified and adapted to VAM.

- **Afs_client** - Client interface to the Amoeba Filesystem Server (AfS).
- **Afs_cache** - a data cache implementation used by both the AFS and DNS server.
- **Afs_common** - types and structures common to server and client.
- **Afs_server** - core of the AFS server.
- **Afs_server_rpc** - the RPC server loop.
- **Amoeba** - the Amoeba core library implementing basic concepts like capabilities.
- **Ar** - ASCII representation of Amoeba structures like capabilities.
- **Arg** - Parsing of command line arguments. *[OCAML305]*
- **Array** - Array operations. *[OCAML305]*
- **Bootdir** - support for Amoebas bootdirectory system.
- **Bootdisk_common** - Kernel Boot directory and disk module server implementation. This server module emulates a DNS/AFS interface for the kernel boot partition holding kernels, binaries needed for bootstrap purposes and configuation files with a very simple filesystem.
- **Bootdisk_server** - the core module of the server.
- **Bootdisk_server_rpc** - the RPC server loop.
- **Bstream** - Generic Bytestream interface
- **Buf** - Provides machine independent storing and extracting of Amoeba structures in and from buffers with bound checking.
- **Buffer** - Extensible buffers. *[OCAML305]*
- **Bytebuf** - Low level Buffer management. Used for example by the Rpc module.
- **Cache** - Provides a fixed table cache.
- **Callback** - Registering Caml values with the C runtime for later callbacks. *[OCAML305]*
- **Cap_env** - support for Amoebas capability environment, similar to UNIX string environment.
- **Capset** - capability sets
- **Circbuf** - Support for circular Amoeba buffers with builtin synchronization primitves needed in a multithreaded program environment.

- **Char** - Character operations. *[OCAML305]*
- **Db** - debug support.
- **Dblist** - double linked lists.
- **Des48** - Crypthograpic de- and encoding.
- **Digest** - Message digest (MD5). *[OCAML305]*
- **Dir** - High level directory service interface.
- **Disk_client** - Virtual Disk Server client interface which provides unique low level access to logical (partitions) and physical disks.
- **Disk_common** - common part used by server and client.
- **Disk_pc86** - i386 dependent parts.
- **Disk_server** - the disk server, running outside the kernel (UNIX).
- **Disk_server_rpc** - the server loop.
- **Dns_client** - Directory and Name service (DNS) client interface
- **Dns_common** - types and structures of the DNS common for client and server modules.
- **Dns_server** - the core module of the Directory and Name server DNS.
- **Dns_server_rpc** - the server loop.
- **Filename** - Filename handling. *[OCAML305]*
- **Format** - Pretty printing. *[OCAML305]*
- **Gc** - Memory management control and statistics; finalised values. *[OCAML305]*
- **Genlex** - A generic lexical analyzer. *[OCAML305]*
- **Hashtbl** - Hash tables and hash functions. *[OCAML305]*
- **Imagerpc** - Image transfer utils.
- **Int32** - 32-bit integers. *[OCAML305]*
- **Int64** - 64-bit integers. *[OCAML305]*
- **Ksys** - Kernel system client interface.
- **Ktrace** - Kernel trace and debug support.
- **Layz** - Deferred computations. *[OCAML305]*
- **Lexing** - The run-time library for lexers generated by [ocamllex]. *[OCAML305]*
- **List** - Single linked List operations. *[OCAML305]*
- **Machtype** - Machine type representation, similar to OCaML's int32 and int64 module, but more general. Remember that OCaML integer are only 31/63 bit wide! The last bit is used internally. So, when the bit length must be guarenteed, use THIS module.
- **Map** - Association tables over ordered types. *[OCAML305]*
- **Marshal** - Marshaling of data structures. *[OCAML305]*

- **Monitor** - Server event monitor support.
- **Mutex** - Supports Mutual Exclusion locks.
- **Name** - Amoeba name interface (easy to handle frontend to DNS) support.
- **Om** - the core module of the Object Manager Server (Garbage Collector).
- **Parsing** - The run-time library for parsers generated by [ocamlyacc]. *[OCAML305]*
- **Pervasives** - This module provides the built-in types (numbers, booleans, strings, exceptions, references, lists, arrays, input-output channels, ...) and the basic operations over these types. *[OCAML305]*
- **Printf** - Formatted output functions. *[OCAML305]*
- **Proc** - Amoeba process client interface. Provides functions to execute (native) Amoeba binaries.
- **Queue** - First-in first-out queues. *[OCAML305]*
- **Random** - Pseudo-random number generator (PRNG). *[OCAML305]*
- **Rpc** - the fundamental communication interface.
- **Sema** - Semaphore synchronization support.
- **Set** - Sets over ordered types. *[OCAML305]*
- **Shell** - some utils for shell like programs.
- **Signals** - Adaption of Amoeba signals to VAM.
- **Sort** - Sorting and merging lists. *[OCAML305]*
- **Stack** - Last-in first-out stacks. *[OCAML305]*
- **Stdcom** - this module implements most of the Amoeba standard commands like *std_info*.
- **Stdcom2** - some more.
- **Stderr** - defines Amoeba standard errors.
- **Stream** - Streams and parsers. *[OCAML305]*
- **String** - String operations. *[OCAML305]*
- **Sys** - System independent system interface... *[OCAML305]*
- **Thread** - Amoeba multithreading module.
- **Unix** - interface to the UNIX operating system (similar to C functions).
- **Vamboot** - a core module implementing Amoeba boot services.
- **Virtcirc** - virtual circuits: distributed access of circular buffers.
- **Weak** - Arrays of weak pointers. *[OCAML305]*
- **WX_adjust** - X widget library: Adjustement object. *[Fab99]*
- **WX_bar** - X widget library: Horizontal and vertical basic widget container. *[Fab99]*
- **WX_base** - X widget library: Base object. *[Fab99]*
- **WX_button** - X widget library: Button object. *[Fab99]*

- **WX_dialog** - X widget library: Dialog object. *[Fab99]*
- **WX_display** - X widget library: X display interface. *[Fab99]*
- **WX_filesel** - X widget library: Fileselection menu object. *[Fab99]*
- **WX_image** - X widget library: Generic image widget. Derived from the WX_pixmap class. *[Fab99]*
- **WX_label** - X widget library: Text label object. *[Fab99]*
- **WX_ledit** - X widget library: Text input object. *[Fab99]*
- **WX_object** - X widget library: Basic WX object support. *[Fab99]*
- **WX_popup** - X widget library: Simple popup menu object. *[Fab99]*
- **WX_progbar** - X widget library: Progress/Value bar object.
- **WX_radiobutton** - X widget library: Radio button object. *[Fab99]*
- **WX_root** - X widget library: Parent object for all toplevel windows. *[Fab99]*
- **WX_screen** - X widget library: X interface utils. *[Fab99]*
- **WX_slider** - X widget library: Slider object.
- **WX_table** - X widget library: Table container for WX objects. *[Fab99]*
- **WX_tree** - X widget library: Tree selector object. *[Fab99]*
- **WX_types** - X widget library: Basic types. *[Fab99]*
- **WX_valbar** - X widget library: Value bar object.
- **WX_viewport** - X widget library: Encapsulates WX objects. *[Fab99]*
- **WX_wmtop** - X widget library: X window manager interface. *[Fab99]*
- **X** - the core X11 graphic windows module. Enables direct X11 programming under VAM. Both UNIX sockets and Amoebs Virtual Circuit X11 communication is implemented. *[Fab99]*
- **XGraphics** - machine-independent graphics primitives.
- **Ximage** - Generic image support. *[Fab99]*
- **Xtypes** - basic X11 types and structures. *[Fab99]*

With this incredible amount of VAM and OCaML modules (and there are still many more) several Amoeba servers, administration and util programs are implemented to built a fully functional distributed operating system either on the top of UNIX or a more raw version on the top of the VX-kernel.
VAM Amoeba servers:

- **AFS**: the atomic filesystem server with backends for UNIX (the filesystem is stored in generic UNIX files or harddisk partitions managed by UNIX) and Virtual Disks (the filesystem is stored on harddisks managed by the VX-Kernel). The AFS filesystem consists of an inode partition holding filesystem informations about each file (described by one inode) and the data partition(s) holding the file data indexed by the file inode.
- **DNS**: the directory and name server. There are slightly different versions for UNIX and VX-Amoeba. The DNS system consists of an inode partition which holds directory capabilities and some basic. The directories itself are saved in generic AFS file obecjts.

- **VAMBOOT**: boot services for an initial operating sysetm startup. The boot server is in fact only a ML script using the boot module.
- **VOM**: a garbage collector server. This server is responsible to cleanup servers periodically. For example the fileserver can contain file object not referenced anymore, directly speaking the capability of the file object is lost. The OM server is a ML script, too.
- **VDISK**: a virtual disk server providing a virtual disk interface for UNIX devices.
- **BDISK**: a bootdirectory server using the virtual disk interface either of native Amoeba or local UNIX devices.

VAM administration and util programs:

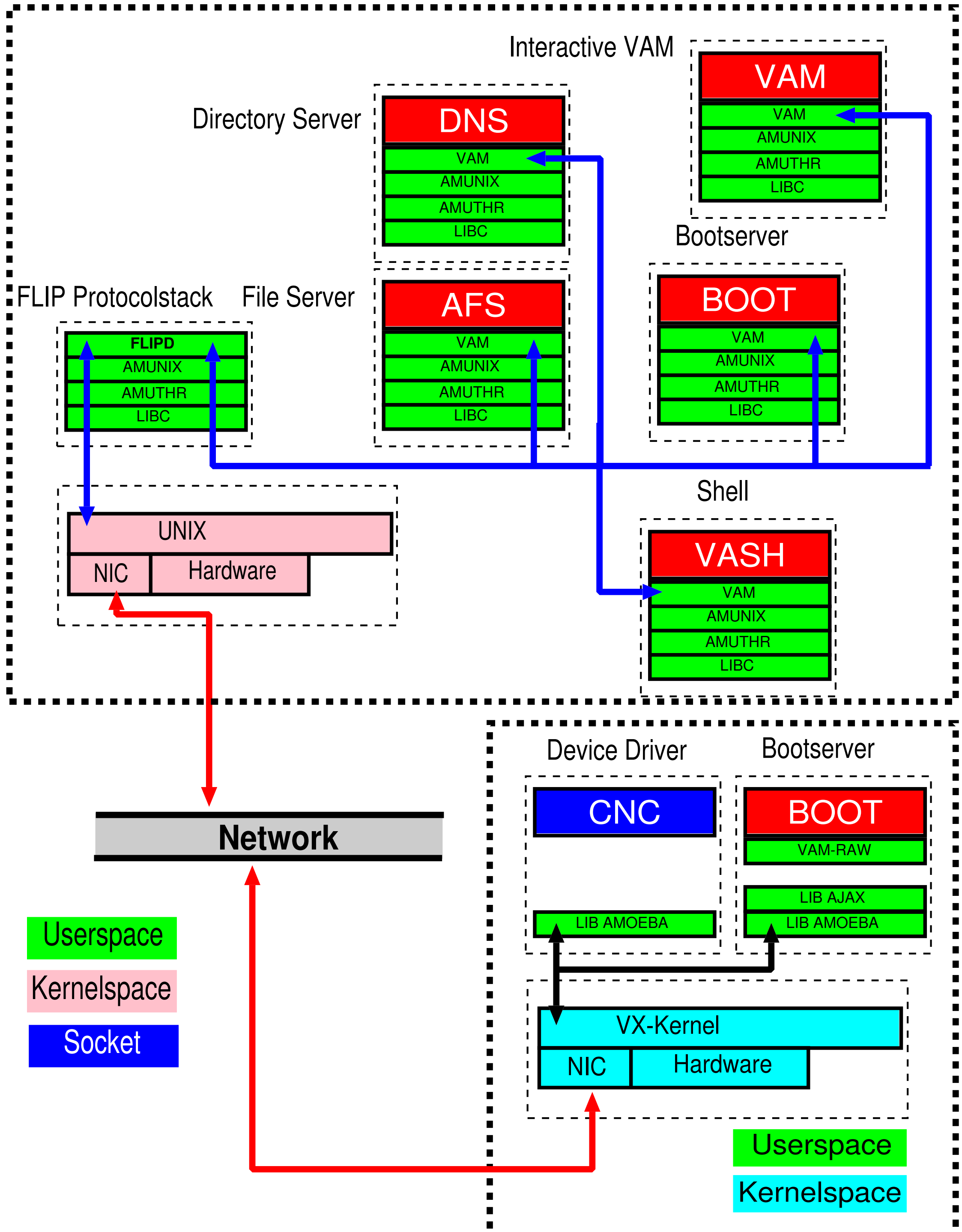
- **vash**: the VAM shell. It's a user interactive command line controlled shell compareable with UNIX bash, providing most of the Amoeba administrative and standard commands.
- **xafs**: a graphical frontend both to the Amoeba directory and filesystem and the UNIX filesystem allowing easy data transfer between both worlds.
- **std**: Amoeba standard commands directly accessible from the UNIX shell.
- **vax**: Executes native Amoeba binaries either located in the Amoeba AFS/DNS system or in the local UNIX filessystem on a specified native Amoeba host. Amoeba kernels can be rebooted with this tool, too.

This is a raw iron version of the virtual Amoeba machine running directly on the top of the VX-Kernel and its bare bone process environment. Only the virtual machine from VAM was adapted to the VX-Kernel. The ML modules kept unchanged. Because the VX-Amoeba process environment has only a restricted UNIX emulation layer, the VAM-UNIX module is restricted in functionality. Only file management is available and some trivial opertaions like the *Unix.time* function. There is no UNIX process control implemented (of course - this makes no sense with native raw Amoeba). All the Amoeba related and generic modules are fully functional.

The VAMNET system forges all previously shown single parts to one hybride operating system:

- The VAM runtime environment with system servers and user interaction,
- the native VX-Kernel and a process envrionment on the top of the VX-Kernel,
- native VX-Amoeba programs, which can be user customized programs.

The next figure 13 shows an example configuration of such a hybrid system. Here, the VAM system is used to control a CNC milling machine connected to external embedded PC104 hardware, running with the VX-Kernel and a CNC machine device driver controlling the axis motion of the machine directly.
First, a boot script will start some basic servers needed for an operational Amoeba system. This is the fileserver AFS *afs_unix* and the directory and name server DNS *dns_unix*. Both store informations in generic UNIX files. Under UNIX, the FLIP server *flipd* is needed for client-server communication, too.
Now the user can start some utils programs, like the VAM shell *vash*. For development purposes, the interactive *vam* program can be used. With this program it's for example possible to compile and execute ML scripts. Also it porvides an online help system containg the VAMNET book.
On the native Amobe side using an embedded PC104 system, there is a boot server to startup the device driver needed to control the connected milling machine. Both computers are connected with 100MBit/s ethernet.
All shown components are merged to one operating system environment. With the VAM shell *vash* it's possible to get access to the native Amoeba Kernel, for example kernel statistic informations can be simply accessed by calling the builtin `kstat` command. The administration of such a hybrid system is quite simple. After the Amoeba file- and directory system was created (using the above shown servers, too), only some capabilities must be inserted in the UNIX environment (using generic UNIX environment variables like `ROOTCAP` specifying the root capability) and the new created directory system, and some system directories expected by various servers and util programs.
Most VAM programs can be executed directly on the native VX-kernel. Only the system Amoeba *libam* and a limited UNIX emulation library *libakjax* are required to implement the VAM virtual machine. This is the only VAM part which must be adapted to the VX-Amoeba process environment. The VAM bytecode executables can be used unchanged for both the native and the AMUNIX Amoeba environment.



(Fig. 13) A VAMNET example configuration connecting a UNIX desktop computer with a network coupled PC104 controller.

One of the incredible results of the VAM project is the fact that the Amoeba emulation layer AMUNIX with the user process implementation of the protocol stack FLIP and the virtual machine approach have only a slightly decreased performance compared with the native VX-Kernel and Amoeba implementation. The following tables gives an impression of the performance and capabilities of the native VX-Kernel system, the AMUNIX and the VAM on the top of AMUNIX system.
The main indicator for the performance of a distributed operating system is the performance of the messaging system, that means the data transfer rate and latency of messages without content (only the message header is transferred).

(TAB. 1) RPC TEST: REMOTE WITH NATIVE VX-AMOEBA KERNEL

Machine configuration	**Transfer direction**	**Transfer rate**	**Latency**
1: AMD-Duron 650 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet	$1 \Rightarrow 2$	11,2 MBytes/s	130 μ s
2: Celeron 700 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet	$2 \Rightarrow 1$	10,6 MBytes/s	136 μ s

(TAB. 2) RPC TEST: REMOTE WITH NATIVE VX-AMOEBA KERNEL

Machine configuration	**Transfer direction**	**Transfer rate**	**Latency**
1: AMD-Duron 650 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet	$1 \Rightarrow 2$	10,5 MBytes/s	170 μ s
2: Cyrix 100 MHz CPU, 32MB RAM, 3COM905 100MBit/s Ethernet	$2 \Rightarrow 1$	9,54 MBytes/s	170 μ s

(TAB. 3) RPC TEST: REMOTE WITH NATIVE VX-KERNEL AND AMUNIX			
Machine configuration	**Transfer direction**	**Transfer rate**	**Latency**
1: AMD-Duron 650 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet, **VX-Kernel**	1 $\Rightarrow$ 2	8,7 MBytes/s	270 μ s
2: Celeron 700 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet, **FreeBSD** $\oplus$ **AMUNIX**	2 $\Rightarrow$ 1	9,1 MBytes/s	260 μ s

(TAB. 4) RPC TEST: REMOTE WITH VX-KERNL AND AMUNIX AND VAM			
Machine configuration	**Transfer direction**	**Transfer rate**	**Latency**
1: AMD-Duron 650 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet, **VX-Kernel**	1 $\Rightarrow$ 2	8,7 MBytes/s	300 μ s
2: Celeron 700 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet, **FreeBSD** $\oplus$ **AMUNIX** $\oplus$ **VAM**	2 $\Rightarrow$ 1	8,7 MBytes/s	300 μ s

The above measurements are example measurements with an accuracy of about $\pm$ 10%. Of course, table 1 shows that the transfer performance of a RPC message transfer from one to another machine reaches it's maximal value. Not only compared with the following AMUNIX and VAM system, also compared with the maximal possible physical transfer rate of 100MBit/s ethernet: 11,9 MBytes/s. This result shows the optiomal adaption of the FLIP protocol stack and the underlying ethernet device drivers to this network system. Table 2 shows results with a different machine 2: a very old Pentium like CPU (Cyrix MMX) with only 100 MHZ core frequency. The VX-Kernel yields to good performance results downto i486 CPU machines.
Using the AMUNIX layer communicating with a native VX-Kernel (Table 3), only a slight decrease in performance and latency can be observed. The transfer rates decreases about 20%, and the latency increased about 100%. With additonal VAM (Table 4), there is no significant difference. This result shows the suitability of ML programming and virtual machine concepts for client-server implementations.
The RPC message passing is not only used for the remote case, but for the local case, too. The following table shows results for the various environments.

(TAB. 5) RPC TEST: LOCAL CASE			
Machine configuration	**Transfer direction**	**Transfer rate**	**Latency**
1: AMD-Duron 650 MHz CPU, 64MB RAM, **VX-Kernel**	$1 \Rightarrow 1$	136 MBytes/s	12 μ s
1: Celeron 700 MHz CPU, 128MB RAM, **FreeBSD** $\oplus$ **AMUNIX**	$1 \Rightarrow 1$	26 MBytes/s	275 μ s
1: Celeron 700 MHz CPU, 128MB RAM, **FreeBSD** $\oplus$ **AMUNIX** $\oplus$ **VAM**	$1 \Rightarrow 1$	22,4 MBytes/s	400 μ s

No surprise the native VX-kernel is the winner. But the AMUNIX and VAM system have sufficient transfer rates and latency times to implement efficient local RPC communication.

FLIP

This section gives an indepth view and details about the implemantation and functionality of FLIP (Fast Local Intranet Protocol) , Amoeba's network protocol used for example by the RPC remote communication and the group communication system.
Some features of FLIP *[KAS93]*:

1. FLIP identifies entities with a location independent 64 bit identifier. An entity can, for example, be a process.
2. FLIP uses a one way mapping between the 'private' address, used to register an endpoint of a network connection, and the 'public' address used to advertise the endpoint.
3. FLIP routes messages based on the 64 bit identifier.
4. FLIP discovers routes on demand.
5. FLIP uses a bit in the message header to request transmission of sensitive messages across trusted networks.

(Reference: *[KAS93]*)
FLIP is a connectionless protocol that is designed to support transparency, efficient RPC, group communication, secure communication, and easy network management. Communication takes place between Network Service Access Points (NSAPs), which are addressed by 64 bit numbers.
NSAPs are location independent, and can move from one node to another (possibly on different physical networks), taking their addresses with them. Nodes on an internetwork can have more than one NSAP, typically one or more for each entity (e.g., process). FLIP ensures that this is transparent to its users. FLIP messages are transmitted unreliably between NSAPs and may be lost, damaged, or reordered. The maximum size of a FLIP message is 2^{32} -1 bytes. As with many other protocols, if a message is too large for a particular network, it will be fragmented into smaller chunks, called fragments. A fragment typically fits in a single network packet.
The reverse operation, reassembly, is (theoretically) possible, but receiving entities have to be able to deal with fragmented messages. The address space for NSAPs is subdivided into 256 56 bit address spaces, requiring 64 bits in all. The null address is reserved as the broadcast address.
The entities choose their own NSAP addresses at random (i.e., stochastically) from the standard space for four reasons. First, it makes it exceedingly improbable that an address is already in use by another, independent NSAP, providing a very high probability of uniqueness. (The probability of two NSAPs generating the same address is much lower than the probability of a person configuring two machines with the same address by accident.) Second, if an entity crashes and restarts, it chooses a new NSAP address, avoiding problems with distinguishing reincarnations (which, for example, is needed to implement atmostonce RPC semantics). Third, forging an address is hard, which, as we will see, is useful for security. Finally, an NSAP address is location independent, and a migrating entity can use the same address on a new processor as on the old one.
A FLIP box consists of an host interface, packet switch, and network interfaces. packets between physical networks, and between the host and the networks. It maintains a dynamic hint cache mapping NSAP addresses on datalink addresses, called the routing table, which it uses for routing fragments. As far as the packet switch is concerned, the attached host is just another network. The host interface module provides the interface between the FLIP box and the attached host (if any). A FLIP box with one physical network and an interface module can be viewed as a traditional network interface. A FLIP box with more than one physical network and no interface module is a router in the traditional sense.

(Reference: *[KAS93]*)
In principle, the interface between a host and a FLIP box can be independent of the FLIP protocol, but for efficiency and simplicity, the interface is based on the FLIP protocol itself. The interface consists of seven downcalls (for outgoing traffic) and two upcalls (for incoming traffic). An entity allocates an entry in the interface by calling *flip_init*. The call allocates an entry in a table and stores the pointers for the two upcalls in this table. Furthermore, it stores an identifier used by higher layers. An allocated interface is removed by calling *flip_end*.
By calling *flip_register* one or more times, an entity registers NSAP addresses with the interface. An entity can register more than one address with the interface (e.g., its own address to receive messages directed to the entity itself and the null address to receive broadcast messages). The address specified, the Private Address, is not the (public) address that is used by another entity as the destination of a FLIP message. However, public and private addresses are related using the following function on the loworder 56 bits:

`Public-Address` = **One-Way-Encryption** `(Private-Address)`

The One-Way-Encryption function generates the Public-Address from the Private-Address in such a way that one cannot deduce the Private-Address from the Public-Address. Entities that know the (public) address of an NSAP (because they have com<AD> municated with it) are not able to receive messages on that address, because they do not know the corresponding private address. Because of the special function of the null address, the following property is needed:

One-Way-Encryption `(Address)=0 if and only if Address=0`

The One-Way-Encryption function is currently defined using DES [National Bureau of Standards 1977]. If the 56 lower bits of the Private-Address are null, the Public-Address is defined to be null as well. The null address is used for broadcasting, and need not be encrypted. Otherwise, the 56 lower bits of the Private-Address are used as a DES key to crypt a 64 bit null block. If the result happens to be null, the result is again encrypted, effectively swapping the result of the encrypted null address with the encrypted address that results in the null address. The remaining 8 bits of the Private-Address, concatenated with the 56 lower bits of the result, form the Public-Address.
Flip_register encrypts a Private-Address and stores the corresponding Public-Address in the routing table of the packet switch. A special flag in the entry of the routing table signifies that the address is local, and may not be removed (as we will see in Section 5). A small EP identifier (End Point Identifier) for the entry is returned. Calling *flip_unregister* removes the specified entry from the routing table.
There are three calls to send an arbitrary length message to a Public-Address. They differ in the number of destinations to which msg is sent. None of them guarantee delivery. *Flip_unicast* tries to send a message point-to-point to one NSAP. *Flip_multicast* tries to send a message to at least ndst NSAPs. *Flip_broadcast* tries to send a message to all NSAPs within a virtual distance hopcnt. If a message is passed to the interface, the interface first checks if the destination address is present in the routing table and if it thinks enough NSAPs are listening to the destination address. If so, the interface prepends a FLIP header to the message and sends it off. Otherwise, the interface tries to locate the destination address by broadcasting a LOCATE message, as explained in the next section.
If sufficient NSAPs have responded to the LOCATE message, the message is sent away. If not, the upcall notdeliver will be called to inform the entity that the destination could not be located. When calling one of the send routines, an entity can also set a bit in flags that specifies that the destination address should be located, even if it is in the routing table. This can be useful, for example, if the RPC layer already knows that the destination NSAP has moved. Using the flags parameter the user can

also specify that security is necessary.
When a fragment of a message arrives at the interface, it is passed to the appropriate entity using the upcall receive. This interface delivers the bare bones services that are needed to build higher level protocols, such as RPC. Given the current low errorrates of networks, we decided not to guarantee reliable communication at the network level, to avoid duplication of work at higher levels [Saltzer et al. 1986]. Higher level protocols, such as RPC, send acknowledgement messages anyway, so given the fact that networks are very reliable it is a waste of bandwidth to send acknowledgement messages at the FLIP level as well. Furthermore, users will never call the interface directly, but use RPC or group communication.

flip_init(ident, receive(), notdeliver(), removeaddr()) ⇒ ifno
: Allocate an entry in the interface.
notdeliver(): A FLIP packet returns; the path to the destination is unknown or the destination is dead. If fl is zero, the flip interface could not locate the destination and we can assume that the destination is dead (we don't have process migration yet). If FLIP_NOTHERE is set in fl, we could try to send the packet again. However, we let the thread who sent the message in the first place do the work (rpc_notdeliver is probably called from the interrupt routine).
removeaddr(): Upcall from the FLIP packet switch to invalidate all cache entries with specified destination address.

flip_end(ifno)
: Close an entry in the interface.

flip_register(ifno, Private-Address) ⇒ EP
: Listen to address.

flip_unregister(ifno, EP)
: Remove address.

flip_unicast(ifno, msg, flags, dst, EP, length)
: Send a message *msg* to *dst*.

flip_multicast(ifno, msg, flags, dst, EP, length, ndst)
: Send a multicast message *msg* to *dst*.

flip_broadcast(ifno, msg, EP, length, hopcnt)
: Broascast a message *msg* upto *hopcnt* hops.

receive(ident, fragment, description)
: Fragment received.

notdeliver(ident, fragment, description)
: Undelivered fragment received.

A FLIP box implements unreliable message communication between NSAPs by exchanging FLIP fragments and by updating the routing table when a fragment arrives.

Similar to fragments in many other protocols, a FLIP fragment is made up of two parts: the FLIP header and the data. A header consists of a 40 byte fixed part and a variable part. The fixed part of the header contains general information about the fragment. The Actual Hop Count contains the weight of the path from the source. It is incremented at each FLIP box with the weight of the network over which the fragment will be routed. If the Actual Hop Count exceeds the Maximum Hop Count, the fragment will be discarded. The Reserved (Res.) field is reserved for future use.

General Format of a FLIP fragment

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Max HopCnt		Actual HopCnt		Res.	Res.	Type	Vers.
0	Destination Address						
0	Source Address						
Length				Message Identifier			
Total Length				Offset			
Total Length							

The Flags field contains administrative information about the fragment. Bits 0, 1, and 2 are specified by the sender. If bit 0 is set in Flags, the integer fields (hop counts, lengths, Message Identifier, Offset) are encoded in big endian (most significant byte first), otherwise in little endian [Cohen 1981]. If bit 1 is set in Flags, there is an additional section right after the header. This Variable Part contains parameters that may be used as hints to improve routing, end-to-end flow control, encryption, or other, but is never necessary for the correct working of the protocol. Bit 2 indicates that the fragment must not be routed over untrusted networks. If fragments only travel over trusted networks, the contents need not be encrypted. Each system administrator can switch his own network interfaces from trusted to untrusted or the other way around.
Bits 4 and 5 are set by the FLIP boxes (but never cleared). Bit 4 is set if a frag ment that is not to be routed over untrusted networks (bit 2 is set) is returned because no trusted network was available for transmission. Bit 5 is set if a fragment was routed over an untrusted network (this can only happen if the Security bit, bit 2, was not set). Using bits 2, 4, and 5 in the Flags field, FLIP can efficiently send messages over trusted networks, because it knows that encryption of messages is not needed.
The Type field in the FLIP header describes which of the (seven) messages types this is (see below). The Version field describes the version of the FLIP protocol; the version described here is 1. The Destination Address and the Source Address are addresses from the standard space and identify, respectively, the destination and source NSAPs. The null Destination Address is the broadcast address; it maps to all addresses. The Length field describes the total length in bytes of the fragment excluding the FLIP header. The Message Identifier is used to keep multiple fragments of a message together, as well as to identify retransmissions if necessary. Total Length is the total length in bytes of the message of which this fragment is a part, with Offset the byte offset in the message. If the message fits in a single fragment, Total length is equal to Length and Offset is equal to zero.
The Variable Part consists of the number of bytes in the Variable Part and a list of parameters. The parameters are coded as byte (octet) strings as follows:

Byte 0:Code, 1:Size, ... Size+1

The (non-zero) Code field gives the type of the parameter. The Size field giv es the size of the data in this parameter. Parameters are concatenated to form the complete Variable Part. The total length of the Variable Part must be a multiple of four byte s, if necessary by padding with null bytes.

FLIP message types

LOCATE
Find network location of NSAP.

HEREIS
Reply on LOCATE.

UNIDATA
Send a fragment point-to-point.

MULTIDATA
Multicast a fragment.

NOTHERE
Destination NSAP is unknown.

UNTRUSTED
Destination NSAP cannot be reached over trusted networks.

GONE
Broadcast with maximal hopcnt that a NSAP is gone (died). This message is used to inform other FLIP boxes around that they can remove this NSAP from their caches (both NSAP-to-Network-Address and RPC-to-NSAP caches).

The basic function of the FLIP protocol is to route an arbitrary length message from the source NSAP to the destination NSAP. In an internetwork, destinations are reachable through any one of several routes. Some of these routes may be more desirable than others. For example, some of them may be faster, or more secure, than oth ers. To be able to select a route, each FLIP box has information about the networks it is connected to.
In the current implementation of FLIP, the routing information of each network connected to the FLIP box is coded in a network weight and a secure flag. A low network weight means that the network is desirable to forward a fragment on. The network weight can be based, for example, on the physical properties of the network such as bandwidth and delay. Each time a fragment makes a hop from one FLIP box to another FLIP box its Actual Hop Count is increased with the weight of the network over which it is routed (or it is discarded if its Actual Hop Count becomes greater than its Maximum Hop Count). A more sophisticated network weight can be based on the type of the fragment, which may be described in the Variable Part of the header. The secure flag indicates whether sensitive data can be sent unencrypted over the network or not.
At each FLIP box a message is routed using information stored in the routing table. The routing table is a cache of hints of the form:

(Address, Network, Location, Hop Count, Trusted, Age, Local)

Address identifies one or more NSAPs. Network is the hardware dependent network interface on which *Address* can be reached (e.g., Ethernet interface). *Location* is the datalink address of the next hop (e.g., the Ethernet address of the next hop). *Hop Count* is a misnomer, but it is maintained for historical reasons.
Count is the weight of the route to *Address*. *Trusted* indicates whether this is a secure route towards the destination, that is, sensitive data can be transmitted unencrypted. *Age* gives the age of the tuple, which is periodically increased by the FLIP box. Each time a fragment from *Adress* is received, the Age field is set to 0. *Local* indicates if the address is registered locally by the host interface. If the *Age* field reaches a certain value and the address is not local, the entry is removed. This allows the routing table to forget routes and to accommodate network topology changes. The *Age* field is also used to decide which entries can be purged, if the routing table fills up.
The FLIP protocol makes it possible for routing tables to automatically adapt to changes in the network topology. The protocol is based on seven message types (see lisiting above). If a host wants to send a message to a FLIP address that is not in its routing table, it tries to locate the destination by broadcasting a LOCATE message#. LOCATE messages are propagated to all FLIP boxes until the Actual Hop Count becomes larger than the Maximum Hop Count. If a FLIP box has the destination address in its routing table, it sends back an HEREIS message in response to the LOCATE.
User data is transmitted in UNIDATA or in MULTIDATA messages. UNIDATA messages are used for point-to-point communication and are forwarded through one route to the destination. MULTIDATA messages are used for multicast communication and are forwarded through routes to all the destinations. If a network supports a multicast facility, FLIP will send one message for all destinations that are located on the same network. Otherwise, it will make a copy for each location in the routing table and send point-to-point messages.
If a FLIP box receives a UNIDATA message with an unknown destination, it turns the message into a NOTHERE message and sends it back to the source. If a FLIP box receives a UNIDATA message that should not be routed over untrusted networks (as indicated by the Security bit), and that cannot be routed over trusted networks, it turns the message into an UNTRUSTED message and sends it back to the source just like a NOTHERE message. Moreover, it sets the Unreachable bit in the message (regardless of its current value). For a message of any other type, including a MULTIDATA message, if the Security bit is set, and the message cannot be routed over trusted networks, it is simply dropped.
If, for a NOTHERE or a UNTRUSTED message, a FLIP box on the way back knows an alternative

route, it turns the message back into a UNIDATA message and sends it along the alternative route. If, for a NOTHERE message, no FLIP box knows an alternative route, the message is returned to the source NSAP and each FLIP box removes information about this route from the routing table.

LOCATE messages must be used with care. They should be started with a Max imum Hop Count of one, and incremented each time a new locate is done. This limits the volume of broadcasts needed to locate the destination. Even though the hop counts are a powerful mechanism for locating a destination and for finding the best route, if routing tables become inconsistent, LOCATE messages may flood the internetwork (e.g., if a loop exists in the information stored in the routing tables in the internetwork). To avoid this situation, each FLIP box maintains, in addition to its routing table, a cache of *(Source Address, Message Identifier, Offset, Destination Network, Location)* tuples, with a standard timeout on each entry. For each received broadcast message, after updating the routing table, it checks whether the tuple is already in the cache. If not, it is stored there. Otherwise, the timeout is reset and the message is discarded. This avoids broadcast messages flooding the network if there is a loop in the network topology.

AMOEBA

This section describes the parts of the C programming interface of Amoeba common to both the native Amoeba (VX-Kernel) and the AMUNIX execution environment. The content of this section has mainly its origin in the Amoeba-5.3 programmer manual provide by the Vrije-University *[AMPRO]* .

AMUNIX

This section deals with details about the Amoeba addon layer for UNIX operating systems. It consists mainly of these parts:

1. The AMUTHR library implementing Amoeba threads in UNIX user process space,
2. the AMUNIX Amoeba library, the core Amoeba library for the AMUNIX environment,
3. the UNIX implementation of the Amoeba protocol stack FLIP, entirely implemented in UNIX user process space,
4. some util programs,
5. and finally last but not least the AMUNIX development environment which enables the build of AMUNIX executables and libraries:

   - Amoeba source tree,
   - a build tree with Amakefiles to built AMUNIX itself.

The AMUNIX system is both a runtime execution and development environment.

AMCROSS

This section deals with details about the Amoeba crosscompiling environment for UNIX operating systems. It consists mainly of these parts:

1. The Amoeba source code tree (shared with AMUNIX),
2. the Amake configuration manager,
3. a build tree with Amakefiles,
4. and finally last but not least the gcc crosscompiler. This gcc was derived from the original source code and can be compiled independently from the Amcross environment.

VX-Kernel Programming

This section explains how to program device drivers and other parts inside the kernel and gives some details about the kernel structure.

The VX-Amoeba scheduler uses a two stage - 3 level priority scheme:

1. **Process priorities**:

   HIGH
   Highest process priority. The kernel is treated like any other process and has this priority. User processes with this highest priority should only be device drivers or other kernel outsourced stuff.

   NORM
   Normal (default) process priority.

   LOW
   Lowest (=background) process priority

2. Each process has it's own 3 level **thread priority** queues:

   HIGH
   Highest possible thread priority.

   NORM
   Normal (default) thread priority (both user and kernelprocess).

   LOW
   Lowest possible thread priority.

   NILT
   The idle thread (only kernelprocess).

Threads in higher priority processes (even they have lowest thread priority) have always higher priority then threads from lower priority processes (no matter their thread priority is).
Kernel threads are always switched non-preemptive. Processes are switched preemptive. User process preemption can be enabled to switch threads of this process with preemption.
The global variable *schedlevel* controlls scheduler activity, too. Each time an hardware interrupt was serviced, and the current (interrupted) thread belongs not to the kernel process (that means the system is in user process mode), the schedule level is checked. In the case, the schedule level is not equal zero (PROC_SCHED), the scheduler is called after an interrupt, for example the timer interrupt. Because interrupts can make threads waiting for an event ready to run, events have a higher priority than normal CPU consuming activities. This leads to the nice feature, that a currenlty CPU consuming thread (maybe regardless of his thread and process priority) will be interrupted for a thread (of maybe another porcess) which was recently woken by an event!
Interrupt are handled with a two level system:

1. Low level interrupt handlers. They are called asynchronously directly due to a pending hardware interrupt. But these functions may not call any global kernel functions directly. Instead, the low level handler queues a high level handler. The low level handler are called in the current thread and process context!
2. High level interrupt handlers. These handlers are then called from the scheduler within a protected thread environment (another kernel thread only living for this purpose). The scheduler

will gain control to this thread immediately after an hardware interrupt occurs as soon as the current running (kernel) thread releases control to the scheduler.

Keep the scheduling policy into mind if you're writing kernel source code. A kernel thread can block the kernel for ever. After a short time (about several seconds), the kernel interrupt hadnler queue will overflow and a kernel panic abort will result. Because only the bare minimum is handled in the (asynchronously called) interrupt handlers, the high level interrupt handler need a chance to execute. Therefore, a kernel function, independent of it job, should execute as fast as possible and release control as early as possible, maybe explicitly with a scheduler call. As described later, the VX-Kernel has a dynamic timer management with arbitrary timer intervalls not limited due to a periodic tick management. But the timer latency and therefore the accuracy is limited by the execution time of the current running kernel thread!

The tables below give an overview of available functions used by various kernel parts and device drivers.

(TAB. 6) **Thread Management [sys/kthread.h]**

Name/Interface	Description
thread_create()	Create a new thread.
thread_exit()	Terminate thread execution.
thread_alloc()	Allocate thread local memory.
thread_switch()	Force a thread switch.
thread_await()	Wait for an event or timeout.
thread_wakeup()	Wakeup thread waiting for event.
thread_await_lock()	Additionally with mutex lock.
thread_id()	Return thread id number.
thread_set_priority()	Set thread priority (HIGH/NORM/LOW).
thread_delay()	Suspend thread for specified time.

(TAB. 7) **Mutex (Mutual exclusion) [module/mutex.h]**

Name/Interface	Description
mu_lock()	Lock a mutex.
mu_trylock()	Try to lock a mutex.
mu_unlock()	Unlock a mutex. Only allowed by owner.

(TAB. 8) **Semaphore [semaphore.h]**	
Name/Interface	Description
sema_init()	Initialize a semaphore variable.
sema_up()	Increment semaphore value.
sema_down()	Decrement sempahore value.
sema_trydown()	Try to decrement semaphore value.
sema_level()	Returns current semaphore level.
sema_mdown()	Decrement semaphore m times.
sema_trymdown()	Try to decrement semaphore m times.
sema_mup()	Increment semaphore m times.

(TAB. 9) **Event [sys/await.h] (aka. thread_XXX)**	
Name/Interface	Description
await()	Wait for an event or timeout.
await_lock()	With additional mutex lock.
await_reason()	Wait for named event (for debugging).
wakeup()	Wakeup waiting threads for an event.

(TAB. 10) **Time(r) Management [sys/timer.h]**	
Name/Interface	Description
timer_set()	Install a new timer handler.
timer_reset()	Change already installed timer handler.
getmilli()	Get current systemtime in milli seconds.
getmicro()	Get current systemtime in micro seconds.
udelay()	Do busy wait delay in micro seconds.

(TAB. 11) **Interrupt Management [sys/kresource.h] [machdep/.../machine.h]**	
Name/Interface	Description
FLAGS	Defines flags variable (macro).
INTR_LOCK	Lock hardware interrupts. Save CPU flags.
INTR_UNLOCK	Unlock hardware interrupts. Restore flags context.
request_irq()	Install and enable interrupt handler.
free_irq()	Release interrupt handler.
probe_irq_on()	Switch IRQ auto probing on.
probe_irq_off()	Switch IRQ auto probing off.

(TAB. 12) **I/O and memory resource Management [sys/kresource.h]**	
Name/Interface	Description
request_mem_region(start,len,name)	Request I/O memory region.
release_mem_region(start,len)	Release I/O memory region.
check_region(start,len)	Check I/O port region.
request_region(satrt,len,name)	Request I/O port region.

(TAB. 13) **I/O Port Management [machdep/.../ioport.h]**	
Name/Interface	Description
in_byte(adr)	Read byte from I/O port.
in_word(adr)	Read 2 bytes from I/O port.
in_long(adr)	Read 4 bytes from I/O port.
out_byte(adr,val)	Write byte to I/O port.
out_word(adr,val)	Write 2 bytes to I/O port.
out_long(adr,val)	Write 4 bytes to I/O port.
ins_byte(adr,ptr,bytecnt)	Read cnt bytes from I/O port to ptr. mem.
ins_word(adr,ptr,bytecnt)	Read cnt bytes from I/O port to ptr. mem.
outs_byte(adr,ptr,bytecnt)	Write cnt bytes to I/O port from ptr. mem.
outs_word(adr,ptr,bytecnt)	Write cnt bytes to I/O port from ptr. mem.

(TAB. 14) **PCI Device Management [pci.h]**	
Name/Interface	Description
pcibios_present()	Test for PCI Bus.
pcibios_find_device()	Find a specified device.
pcibios_find_class()	Find a specified device class.
pcibios_strerror()	Convert error value to string.
pcbios_read_config_byte()	Read PCI device config byte.
pcbios_read_config_word()	Read PCI device config word (16 Bit).
pcbios_read_config_dword()	Read PCI device config double word (32 Bit).
pcibios_write_config_byte()	Write PCI device config byte.
pcibios_write_config_word()	Write PCI device config word (16 Bit).
pcibios_write_config_sword()	Write PCI device config double word (32 Bit).

aalloc
Allocates non-returnable memory on an arbitrary boundary with spezified alignment. Alignment argument must be on power of two. The length is specified in byte units.

getblk
This functions allocates page-aligned memory, in page-sized amounts (a page is a MMU page, 512 bytes or so).

relblk
Return memory gotten from *getblk*.

malloc
Allocates reusable memory. The memory is aligned to *ALIGNMENT* bytes, specified in the *malloctune.h* header file, currently 16 bytes, to satisfy allmost all alignments needed by the kernel. The length is specified in byte units.

free
Release memory previously allocated with *malloc*.

PROGRAMMING INTERFACE *[sys/proto.h] [stdlib.h]*

```
char* aalloc ( vir_bytes size ,
               int align );

vir_bytes getblk ( vir_bytes size );

void relblk ( vir_bytes paddr );

void* malloc ( size_t len );

void free ( void *ptr );
```

IO-Ports of hardware devices can be accessed with the following functions. Make sure the IO regions is previously allocated by the resource management and not used by any other device.
On PC86 machines, the IO-Port address space is in the range of {0x00-0xFFFF}. In contrast to other kernels running on this machine system, the full IO range can be accessed and is managed by the kernel! Be aware: there is no exception raised if a device driver accesses an I/O port which was never requested by the device driver!

PROGRAMMING INTERFACE *[machdep/arch/XXX/ioport.h]*

```
void out_byte ( int _port ,
                int _val );

void out_word ( int _port ,
                int _val );

void out_long ( int _port ,
                int _val );

int in_byte ( int _port );

int in_word ( int _port );

long in_long ( int _port );

void outs_byte ( int _port ,
                 char * _ptr ,
                 int _bytecnt );

void outs_word ( int _port ,
                 char * _ptr ,
                 int _bytecnt );

void outs_word_l ( int _port ,
                   char * _ptr ,
                   int _wordcnt );

void outs_long_l ( int _port ,
                   char * _ptr ,
                   int _longcnt );

void ins_byte ( int _port ,
                char *_ptr ,
                int _bytecnt );

void ins_word ( int _port ,
                char *_ptr ,
                int _bytecnt );
```

```
void ins_word_l ( int _port ,
                  char *_ptr ,
                  int _wordcnt );

void ins_long_l ( int _port ,
                  char *_ptr ,
                  int _longcnt );
```

Bytes of length 8bit, words of length 16bit, and longs (double words) of length 32bit can be read and written with the respective functions declared above. There are some additional functions for old and slow hardware, inserting a short delay: *out_##_p(),in_##_p()*. Memory regions can be copied to and from hardware ports with the *outs_##(),ins_##()* functions. Take care about the count parameter values!

The kernel supports timer management with microsecond resolution (depenending on the resolution of the timer hardware device).
The kernel hardware timer is now implemented with an "one-shot" behaviour with dynamically adapated time intervals. After an interval time passed, an interrupt is triggered, and the interrupt service routine will program the hardware timer with the next desired time interval. This enables the kernel scheduler to handle timeouts with millisecond resolution. With special functions it's possible to realize software timers delivering microsecond resolution. Absolute time values are stored in 64-Bit unsigned integers (unsigned long long).

timer_set
The *timer_set* function initializes a software interval timer. The user supplied function *fun* will be called after the time interval *period* in *unit* (SEC, MILLISEC, MICROSEC) relative to the current system time has elapsed. If the *once* argument is equal zero, the timer function will be called periodically, else only one time.

timer_reset
Change (reset or remove) an already installed timer handler. If the period value is set to zero, the timer will be removed.

hw_set_timer
The *hw_set_timer* function sets the hardware timer to the new interval *usec* with microsecond resolution. Always realtive to the last timer interrupt. Returns the passed time in microseconds. Should only be used internally.

getmicro, getmilli
The *getmicro* and *getmilli* functions return the current system time in micro- or milliseconds. The *passed_micro* function returns the actual microseconds passed since the last timer interrupt.

unit
The time unit used in various functions: `MICROSEC, MILLISEC, SEC, MINUTE, HOUR`.

PROGRAMMING INTERFACE *[sys/timer.h] [amoeba.h] [sys/hw_timer.h]*

```
void  timer_set ( void (*fun)() ,
                  long arg ,
                  interval period ,
                  int unit ,
                  int once );

void  timer_reset ( void (*fun)() ,
                    long arg ,
                    interval period ,
                    int unit ,
                    int once );

unsigned int  hw_set_timer ( uint32 usec );
```

uint32 **passed_micro** (void);

uint64 **getmicro** (void);

uint32 **getmilli** (void);

Both, user and kernel threads have the same programming interface. The thread management module in the VX-Kernel was fully revised and differs interally from the orginal Amoeba kernel, but the programming interface kept nearly unchanged, except some enhancements for therad creation.
This module provides simple thread support. Both, within the kernel, and outside in user progams, the same API is used.

thread_newthread
Creates a new thread. The thread function being called in the new thread context must have the format:

```
void fun(char *param, int paramsize)
```

The *thread_newthread* function will return the thread id number of the new creates thread. The *param* memory must be allocated with *malloc* because it's freed after a thread exit. Obsolete! Use *thread_create* instead.

thread_create
Creates a new thread. The thread function being called in the new thread context must have the format:

```
void fun(long arg)
```

The *thread_newthread* function will return the thread id number of the new creates thread. The *arg* must not be allocated with *malloc* because it's not freed after a thread exit.

thread_switch
Release control from the current thread. The calling thread will be blocked, and the scheduler will choose another thread being runable, if any.

thread_exit
Exit a thread and do cleanup. The memory pointed by *param* will be released!

thread_id, thread_kid
Return the process loacl and kernel global thread id number of the current thread.

thread_await
Wait for event *ev* or timeout within interval *to*. The event variable is only a key value for the kernel, therfore any process address of a global variable can be used for an event await or wakeup. The returned status value is either 0 (got event), or -1 (interrupted/timeout).

thread_await_lock
Wait for event *ev* or timeout within interval *to*. The event variable is only a key value for the kernel, therfore any process address of a global variable can be used for an event await or wakeup.
This version is protected with a mutex lock. The user thread must lock this mutex (and protect his critical section) before calling await_lock. The await_lock syscall in the kernel will unlock this mutex after preparing the thread for event awaiting, but before the scheduler performs a thread switch.

thread_wakeup
Wakeup waiting threads for event *ev*. Returns number of woken threads.

thread_delay
Delays execution of current thread for a given amount of time. The unti of the interval time is specified by the second argument: `MICROSEC, MILLISEC, SEC, MINUTE, HOUR.`

PROGRAMMING INTERFACE *[sys/kthread.h] [sys/timer.h] [amoeba.h]*

```
int thread_newthread ( void (*fun)() ,
                       int stacksize ,
                       char *param ,
                       int paramsize );

int thread_create ( void (*fun)() ,
                    long arg ,
                    int stacksize ,
                    PRIO prio );

int thread_id ( void );

int thread_kid ( void );

int thread_exit ( void );

int thread_await ( event ev ,
                   interval to );

int thread_await_lock ( event ev ,
                        interval to ,
                        mutex *mu );

int thread_wakeup ( event ev );

int thread_delay ( interval to ,
                   int unit );
```

check_region
Call this function before probing for your hardware or accessing any IO ports. The start address and the length of the port region in byte units must be specified.

request_region
Register an IO port region for a device driver. The supplied name string specifies the device.

release_region
Release previously reserved IO port region. Make sure the start address and length are the same as used with the *reuqest_region* function.

PROGRAMMING INTERFACE *[sys/kresource.h]*

```
int  check_region ( unsigned int start ,
                    unsigned int length );

int  request_region ( unsigned int start ,
                      unsigned int length ,
                      char *name );

void  release_region ( unsigned int start ,
                       unsigned int length );
```

request_mem_region
: Register a device memory region for a device driver. The supplied name string specifies the device.

release_mem_region
: Release previously reserved memory region. Make sure the start address and length are the same as used with the *reuqest_region* function.

PROGRAMMING INTERFACE *[sys/kresource.h]*

```
int  request_mem_region ( unsigned long start ,
                          unsigned long length ,
                          char *name );

void  release_region ( unsigned long start ,
                       unsigned long length );
```

The VX-Kernel can be configured before loaded and started with a program like option/flag list:

```
<kernelname> [-optionname:value [-optionname:value]...]
Example:
kernel -noreboot:2 -ide1:1
```

All kernel parameter expect integer values in decimal or hexadecimal format (with a preceeding 0x or 0X format identifier). The value zero indicates a disabled option. The following list shows all currently availble options.

aip
Auto IRQ probing ofr asynchrounus serial interface.

1
enabled

0
disabled

ide0
IDE controller configuration:

1
probe only for master device

2
probe only for slave device

8
Don’t probe for IDE controller 0

ide1
IDE controller configuration:

1
probe only for master device

2
probe only for slave device

8
Don’t probe for IDE controller 1

kbl
Keyboard mapping:

1
: US mapping

2
: German mapping

mem
: Set the RAM memory size in MByte. The BIOS value gives the size only modulus 64 MByte! A value larger then the physical memory will cause a system crash.

noreboot
: Set the reboot mode after a kernel panic:

0
: Auto reboot without halting the system and waiting for user interaction.

1
: No auto reboot. User must press the reset button to reboot the system.

2
: Switch back to VGA text mode (a kernel panic during graphics mode!).

3
: Switch back to VGA text mode using VGA BIOS. The more safe method on accelarted video cards.

rios0
: Define a start address of a reserved IO port region. Usefull for PCI devices with IO addresses (assigned by the BIOS) conflicting with IO addresses of ISA/PC104 devices. IO addresses of PCI devices lying inside the reserved region will be relocated outside.
Up to 10 reserved IO regions can be defined (rios0..rios9).

rioe0
: Define the end address of the above reserved region. If rios $>$ 0, than rioe defaults to the highest IO address (0xFFFF).
Up to 10 reserved IO regions can be defined (rioe0..rioe9).

The operational functionality of the VX-kernel is structured with different servers.

Synopsis

Currently built into the kernel *[AMSYS]*

Description

This server provides Amoeba programs with random numbers. No rights are required to use this server. Random numbers are the privilege of all. There is only one command and that returns a random number of the size specified.

Interface

The programming interface consists of the single command particular to the server (whose name begins with rnd) and the standard server commands (whose names begin with std). A summary of the supported commands is presented in the following two tables. For a complete description of the interface routines see rnd(L) and std(L). Not all the standard commands are supported since they are not all pertinent. Std_copy and std_destroy are not implemented. std_age, std_restrict and std_touch are implemented but simply return STD_OK and do nothing further. They do no error checking.

(TAB. 15) STANDARD FUNCTIONS

Function Name	Required Rights	Error Conditions	Summary
std_age	NONE	RPC Error	Does nothing
std_info	NONE	RPC Error	Returns info string
std_restrict	NONE	RPC Error	Does nothing
std_touch	NONE	RPC Error	Does nothing

(TAB. 16) RND FUNCTIONS

Function Name	Required Rights	Error Conditions	Summary
rnd_getrandom	NONE	RPC Error	Returns a random number of the requested size

Administration

There is only one administrative task relating to the random server. That is installing the capability of one of the random servers as the default random server. The place for installing the default server capability is described in ampolicy.h by the variable DEF_RNDSVR and is typically /system/cap/randomsvr/default . However the place to install it is via the path /super/cap/randomsvr/default which is the public directory. (The /system directory may vary from user to user but typically points to the public version.) It is normal practice to allow the boot server to maintain the default capability.

VX-Kernel: External Device Drivers

It's also possible to build any kind of device drivers in user space. The device driver is handles just as a generic user space process and can be started as any usual process using Amoeba's process interface. The only difference (if at all): the user space device driver needs a special capability to gain control over I/O ports and for requesting interrupts (currently the kernel root capability for simplicity).

Due to the fact that the VX-Amoeba Kernel is a true Microkernel, device drivers can be implemented both inside and ouside of the kernel. Most source code of program sections of a device driver can be shared between kernel inside and ouside implementations. Outside the kernel, device drivers are executed in a normal process context. They need only a special protection capability to access hardware resources.
Main differences between these two implementation methods are exist in the way resources are enabled and used.
Resources are:

IO Ports
: Generic hardware I/O ports of hardware devices.

IRQ
: Interrupts generated by hardware devices.

MEM
: Hardware device memory mapped in kernel address space. Either configuration or data space of a device.

PCI
: Access of devices connected to the local PCI bus system.

TIMER
: Timing services scheduled by the VX-Kernel timer management and the thread/process scheduler. This resource is needed to periodically execute user supplied functions, for example device driver timeout management.

User space I/O port access routines used in process space device drivers. All functions managing these resources are implemented with the local IPC interface. The kernel system resource server is responsible for this task.

On PC86 machines, the IO-Port address space is in the range of {0x00-0xFFFF}. In contrast to other kernels running on this machine system, the full IO range can be accessed and is managed by the kernel!
Be aware: all user processes with mapped I/O ports can access I/O ports used by other user processes without an exception!

io_check_region
Before an external process can access I/O ports, in contrast to device drivers inside the kernel, the process must register and map the desired I/O region. Because only one process can map a specific I/O region, the *io_check_region* function must be called to check the availability of the resources.

io_map_region
After the *io_check_region* function returns the *STD_OK* status, the *io_map_region* function can be used to map and register the I/O port region. After this call, I/O ports can be accessed with the below explained functions.
Warning: the *devname* argument specified with the *io_map_region* function MUST be allocated with the *malloc/alloc* functions, or the process will be terminated with an exception (that means, the devname string must be outside of readonly text segments!).

io_unmap_region
Either implecitely on process exit, or explicitely, the I/O region can be unmapped with the *io_unmap_region* function.

io_vtop
The *io_vtop* function translates a virtual process address region of specified length to the physical (real) address. This is needed for DMA transfers, for example. The memory region can be a mapped hardware segment, too.

io_setpvl
The *io_setpvl* function can be used to gain full I/O access for a program with changing the IOPL.

[syscap]
The system capability of the kernel. Currently the kernel root directory capability. Can be lookuped from the DNS server.

PROGRAMMING INTERFACE *[ioport.h] [sys/iomap.h]*

```
errstat io_check_region ( unsigned int start ,
                          unsigned int size ,
                          capability *syscap );

errstat io_map_region ( unsigned int start ,
                        unsigned int size ,
                        char *devname ,
                        capability *syscap );

errstat io_unmap_region ( unsigned int start ,
```

```
                                    unsigned int size ,
                                    capability *syscap );

errstat  io_vtop ( long vaddr ,
                   long vlen ,
                   long *paddr ,
                   capability *syscap );

errstat  io_setpvl ( int pvl ,
                     capability *syscap );
```

PROGRAMMING INTERFACE *[machdep/arch/XXX/ioport.h]*

```
void out_byte ( int _port ,
                int _val );

void out_word ( int _port ,
                int _val );

void out_long ( int _port ,
                int _val );

int in_byte ( int _port );

int in_word ( int _port );

long in_long ( int _port );

void outs_byte ( int _port ,
                 char * _ptr ,
                 int _bytecnt );

void outs_word ( int _port ,
                 char * _ptr ,
                 int _bytecnt );

void outs_word_l ( int _port ,
                   char * _ptr ,
                   int _wordcnt );

void outs_long_l ( int _port ,
                   char * _ptr ,
                   int _longcnt );

void ins_byte ( int _port ,
                char *_ptr ,
                int _bytecnt );

void ins_word ( int _port ,
                char *_ptr ,
                int _bytecnt );

void ins_word_l ( int _port ,
                  char *_ptr ,
                  int _wordcnt );

void ins_long_l ( int _port ,
                  char *_ptr ,
```

```
int _longcnt );
```

Bytes of length 8bit, words of length 16bit, and longs of length 32bit can be read and written with the same functions declared already in the kernel section. Memory regions can be copied to and from hardware ports with the *outs_##(),ins_##()* functions respectively. Take care about the count parameter!

Here is a short example for accessing I/O ports through user processes.

```
#include < sys/iomap.h >
#include < ioport.h >

#define KERNELSYSCAP "/hosts/dio01"
#define MYNAME "MYSERVER"

int main()
{
   char *devname=malloc(SIZEOF(MYNAME)+1);
   errtstat err;
   capability syscap;

   err = name_lookup (KERNELSYSCAP,&syscap);
   if (err != STD_OK)
   {
   failwith("Can't lookup kernel cap");
   }
   err = io_checkregion (0x278,4,&syscap);

   if (err != STD_OK)
   {
   failwith("I/O region already used");
   }

   strcpy(devname,MYNAME);
   err = io_map_region (0x278,4,devname,&syscap);

   if (err != STD_OK)
   {
   failwith("IO mapping failed...");
   }

   out_byte (0x278,0xff);


   /* all done */

   io_unmap_region (0x278,4,&syscap);

return 0;

}
```

The way interrupts are handled is different inside and ouside the kernel. Inside th kernel, simply an interrupt handler function is installed. Outside the kernel, this method is not preffered, because an interrupt handler executes usually in an arbitrary context of the current running process (kernel or user process). Therefore, interrupt handling differs outside the kernel in a user process. A dedicated interrupt handler thread requests an interrupt from the system server of the kernel. If this succeeds, the interrupt thread calls the *interrupt_await* function signal the await for an interrupt previously registered. If the assigned hardware devuce triggers the interrupt, the kernel interrupt module will wakeup this waiting thread. The interrupt thread will executes as soon as possible, depending on the process and thread priority. After the work is done, or the interrupt source in the case of shared interrupts is not handled by this device driver, the interrupt thread will call the *interrupt_done* function to signal the kernel the finished service for this hardware interrupt. In the case that there is more than one handler (shared interrupt signals), further handler functions will be executed untill one handler signals the succesfull service of the interrupt.
To request and service interrupts, the kernel system capability (currently the kernel root directory capability) is requiered. The kernel system server manages user space interrupt requests. Ths register functions are stubs for IPC message transfers to the system ISR server inside the kernel with the portname `sys::isr-server`. The message request holds the content of the user supplied *isr* structure.

interrupt_register
: Register an interrupt service thread. The following entries in the *isr* structure must be set:

```
isr.irq=[hardware irq number];
isr.flags=[IRQ_NORMAL | IRQ_SHARED];
strcpy(isr.devname,"mydev");
```

Important: This fucntion must be called within the interrupt service thread!

interrupt_await
: Now the ISR handler thread can wait for interrupts. The event variable is taken from *isr* structire previously used with *interrupt_register* :

```
interrupt_await(isr.irq,isr.ev);
```

The interrupt kept locked untill *interrupt_done* is called.

interrupt_done
: After the interrupt service routines finished his work - either the interrupt source was handled or the ISR find out that's not his device that triggers the interrupt - this function MUST be called with the appropiate flag set:

```
status=[IRQ_SERVICED | IRQ_UNKNOWN];
interrupt_done(isr.irq,status);
```

interrupt_unregister
: This function unregisters a previously registered interrupt service routine. The *isr* structrue

must be same as used by the *interrupt_register* function.
Usuallay, interrupt resource of user space device drivers are released on process exit.

PROGRAMMING INTERFACE *[sys/isr.h]*

struct isr_handler { int irq ;
int flags ;
char devname[MAX_DEVNAME_LEN] ;
event ev ;
unsigned long id }

typedef struct isr_handler isr_handler_t
, *isr_handler_p

errstat **interrupt_register** (isr_handler_p isr ,
capability *syscap);

errstat **interrupt_unregister** (isr_handler_p isr ,
capability *syscap);

errstat **interrupt_await** (int irq ,
event ev);

errstat **interrupt_done** (int irq ,
int status);

EXAMPLE

```
capability syscap;
#define HOSTCAP=/hosts/juki01
void myisr()
{
    errstat err;
    int stat;
    isr_handler_t isr;
    err=name_lookup(HOST_PATH,&syscap);
    ...
    isr.irq=3;
    isr.flags=IRQ_NORMAL;
    strcpy(isr.devname,"Serial Comm");
    err=interrupt_register(&isr,&syscap);
    ...
```

```
    for(;;)
    {
    stat=interrupt_await(isr.irq,isr.ev);
    ...
    stat=interrupt_done(isr.irq,IRQ_SERVICED);
    }
}
```

Timer management also exists for user processes with microsecond resolution (depenending on the resolution of the timer hardware device and the dead time of process system calls).
In contrast to the kernel implementation, there is no user supplied timer function called on a timer event. Instead, a await-wakeup mechanism is used, simlar to user process interrutps.

timer_init
: The *timer_init* function initializes and installs a new software interval timer. The user supplied event *ev* will be wakedup after the time interval *period* in *unit* (SEC, MILLISEC, MICROSEC) relative to the current system time has elapsed. If the *once* argument is equal zero, the timer function will be called periodically, else only one time.

timer_reinit
: Change (reset or remove) an already installed timer handler. If the period value is set to zero , the timer handler witll be removed.

timer_await
: The *timer_await* function is blocked untill the timer event was raised by the kernel timer module.

unit
: The time unit used in various functions: `MICROSEC, MILLISEC, SEC, MINUTE, HOUR`.

PROGRAMMING INTERFACE *[sys/utimer.h]*

```
typedef int   timer_event

int timer_init ( timer_event *ev ,
                 interval period ,
                 int unit ,
                 int once );

int timer_reinit ( timer_event *ev ,
                   interval period ,
                   int unit ,
                   int once );

interval timer_await ( timer_event *ev );
```

The IPC module is intended to use by userspace device drivers to communicate locally both with the kernel and other device drivers outside the kernel in a fast and easy way, very similar to the RPC interface used for both local and remote interprocess communicarion.

VAM Development System

The development system contains several ML libraries, the virtual machine, an interactive ML interpreter called *VAM*, the ML compilers (also builtin *VAM*) and many more.
Documentation for the following libraries are available:

The VAM system is both delivered in source code and binary form. The build process from scratch requires only a few steps. The VAM system depends on the following packages:

amake-unix
The Unix version of the Amoeba make program. This program must be compiled first of all other packages. It's only a bootstrap version. The final one is compiled in the following AMUNIX environment.

amoeba-src
The source code of the native Amoeba system and the AMUNIX system.

amoeba-build-amunix
The build tree of the AMUNIX system. This incorporates a reduced Amoeba core library targeting the Unix environment, the UNIX version of the communication protocol stack *flipd*, a thread package, and some util programs.

amoeba-build-crossutils-myos (optional)
Programs and utils for the UNIX crosscompiling envionment. This enviornment enables the building of native Amoeba programs under Unix.

amoeba-build-amcross (optional)
The amoeba build tree using the above mentioned crosscompiling tools. Here, native Amoeba libraries, programs and kernels can be build.

The first three are required and must be build in the order shown above. The steps are described in the AMUNIX manual, not here.

1. You need the bash shell under the path */bin/bash*
2. Make sure your path setting includes the current directory FIRST for all other paths:

```
export PATH=.:$PATH
```

3. Choose the appropriate system dependent Amakefile.sys.<myos> and edit the Amakefile.sys file:

```
vi Amakefile.sys
```

4. Edit at least the path settings in this Amakefile:

```
VAMDIR = /amoeba/vam-1.7;
INSTALLDIR = /amoeba/Vam-1.7;
```

5. Build the system. Simply start from the VAM source top directory:

```
build clean    # remove old remains
build all
build install
```

6. You can always set parts or the all of the compiling environment to the initial clean state with the command:

```
build clean
```

7. You can enter subdirectories of the build tree and build only single parts of the system with the same commands shown above:

```
build clean
```

8. If something goes wrong, contact the author:

```
Dr.  Stefan Bosse
BSSLAB, Bremen, Germany
```

```
http://www.bsslab.de
```

The source distribution:

$VAMDIR
The top level directoty holding the main Amakefiles and the toolset directoty with various common defintions needed to build OCaML and VAM. Default is */amoeba/vam-1.7*.

$VAMDIR/src
The sources.

$VAMDIR/src/amoeba
The basic Amoeba module. Contains the Amoeba basic modules with RPC support and some client modules for accessing Amoeba servers.

$VAMDIR/src/buffer
Generic byte buffer management.

$VAMDIR/src/debug
Some debug management.

$VAMDIR/src/ocaml
The sources for the OCaML system. These sources base currently on the offical 3.05 release, but are strongly modified.

$VAMDIR/src/os
Operating system dependent modules.

$VAMDIR/src/scripts
Various VAM scripts, for example the VAM compiler script *vamc*.

$VAMDIR/src/server
Common server implementations like AFS and DNS.

$VAMDIR/src/system
VAM system programs, for example the AFS and DNS servers.

$VAMDIR/src/termianl
The readline terminal module.

$VAMDIR/src/threads
Thread implementation based on Amoeba threads (AMUNIX version).

$VAMDIR/src/top
The main module for VAM.

$VAMDIR/src/unix
The Unix library.

$VAMDIR/src/xwin
The X-Library fully reqritten in ML (by Fabrice Le Fessant) and a simple widget library build on the top of xlib.

The binary distribution:

$INSTALLDIR
The top level directoty holding the OCaML and the VAM system binaries, libraries and interface files. The default is */amoeba/Vam-1.X*.

$INSTALLDIR/bin
VAM binaries and scripts.

$INSTALLDIR/config
VAM configuration scripts like the boot script *boot* and the virtual object manager *vom*.

$INSTALLDIR/doc
This directory holds several documents about VAM and OCaML..

$INSTALLDIR/interface
VAM interface files.

$INSTALLDIR/lib
VAM ML and C libraries.

$INSTALLDIR/ocamlsys
The underlying OCaML system: binaries, libraries, interface files, headers.

$INSTALLDIR/toolset
VAM development files, for example the Amakfile.sys VAM system configuration and other tool scripts needed by Amoebas make program *Amake* to build VAM applications. There are also default Amakefile templates.

ML source code can be compiled and linked to archives or bytecode programs. Additionally, new custom designed virtual machines and interactive systems can be genrated. There are two ways to perform this tasks:

1. Using the *vamc* script as a frontend to the VAM-ML compiler,
2. or using the Amoeba configuration manager *amake* with an Amakefile defining target clusters and sources in a much more comfortable way than using the *vamc* script.

This is the simplest and easiest way to compile ML source code and built VAM bytecode programs suitable for executing on native Amoeba-VAM (VAMRAW) and UNIX-VAM. But not only ML code can be compiled. This script provides a frontend for the C compiler (but using the ocaml frontend with path settings), too.

Usage

```
vamc [Options] <input> -o <output>
```

Program arguments

-c
Compile a single source code file into object bytecode format. Ths file extension determines the compiler mode: ML or C.

-i
Create an ML interface file from ML source.

-a
Link specified source object files into an archive file specified with the output argument.

-build-vm
Build a new custom designed virtual machine (only UNIX native format).

-g
Add debugging informations. On an uncaught exception during program execution, only sources and objects supplied with debug informations are printed in the function backtrace. All compiling steps must be provided with this option, also the link process.

-usepp
Use the C preprocessor. The VAM-ML compiler supports C style like preprocessor directives within ML source code!

Examples

```
vamc -c myml.ml
vamc -o mybyteprog myml.cmo
vamc -a -o myarch.cma myml.cmo
vamc -c main.ml
vamc -o mabyte2 main.cmo myarch.cma
```

The *amake* program provided in the AMUNIX binary collection can also be used to built bytecode programs or libraries. Amake needs a list of source code, a definition of tools to transform source of a specified type, for example ML or C code, into another type, for example object code or archives. But object code can be again a source for a so called cluster. The clutser defines the way to transform source into target codes. Several clusters in a chain are needed to built programs. To make live easy, the tools and source-to-target transformations are already provided by VAM and AMUNIX.
One of the features is that the source directory is separated from the build directory. The source directory must contain a file specifying all the sources, for example:

```
CNC_LIB = {
    $PWD/help.ml,
    $PWD/cnc_defaults.ml,
    $PWD/dxf.ml,
    $PWD/dxf.mli,
    $PWD/cnc.ml,
};

CNC_MAIN = {
    $PWD/main.ml
};
```

Now make a build directory somewhere else. In this directory place the Amakefile:

```
TOP=/amoeba/Vam-1.7;
SRC_ROOT=/home/sbosse/proj/dxf2cnc/src;

%include $TOP/Amakefile.sys;
%include $TOP/Amakefile.common;

%include $SRC_ROOT/Amake.srclist;

INCLUDES = {
    -I,
    $SRC_ROOT,
};

# some general compile defintions
DEFINES = {
    -g,
};

# the sources with source specific compile flags
SRC = {
    $CNC_LIB # [flags={-po,-DLOG}]
};

# the target library
LIB = dxf2cnc.cma;
```

```
%include $VAMDIR/toolset/ocaml.lib;
%include $VAMDIR/toolset/ocaml.sys;

%instance libcluster ($LIB,$SRC);

# and the bytecode program
PNAME = dxf2cnc;
PLIBS = {
   unix.cma,
   str.cma,
   dxf2cnc.cma,
};

PSRC = $CNC_MAIN;
VM = $VAMDIR/bin/vamrun;
%instance bytecode-exe ($PNAME,$PLIBS,$VM,$PSRC);
```

This Amakefile must first specify the path to the binary VAM tree (the toplevel path variable `TOP`) and the path to the source code your are want to compile. The Amakefile now includes two important files describing system dependent and independent features of the VAM system. They are already adjusted to your system. Some more Amake variables will define the sources and targets. There are two main cluster, `libcluster` and `bytecode-exe`, which will build the intermediate library `dxf2cnc.cma`, and finally the bytecode program `dxf2cnc`. These two clusters must be build with independent amake calls:

```
amake dxf2cnc.cma
amake dxf2cnc
```

Note: It's important to create the directory `interface` in the build directory!
In contrast to the *vamc* script, you must specify VAM system libraries which are needed by your program. They must be added to the library list in the given order:

- `debug.cma`
- `buffer.cma`
- `unix.cma`
- `str.cma`
- `threads.cma`
- `os.cma`
- `test.cma`
- `amoeba.cma`
- `server.cma`

In the case, the graphical widget system is used, there are some more:

- `xlib.cma`
- `wxlib.cma`

Content of the Amoeba library, building the fundamentals of the VAM system:

This is the base Amoeba module covering the following areas:

- Basic types
- Capability and Port functions
- The RPC header structure
- Encryption

Basic Types

The following types are used in capabilities and headers:

PROGRAMMING INTERFACE

type rights_bits = Rights_bits of int

type obj_num = Objnum of int

type command = Command of int

type errstat = Errstat of int

type status = Status of int

type port = Port of string

type privat = { **mutable** prv_object: obj_num ;
mutable prv_rights: rights_bits ;
mutable prv_random: port }

The main Amoeba structure is the *capability*. The main purpose of a *capability* is to give an arbitrary object an unique identifier. Objects can be of several types:

- Files
- Directories
- Processes
- Memory Segments
- Server

and many more. The RPC header structure is needed for communication between clients and servers. Details on RPC programming are shown in the Tutorial (??).

PROGRAMMING INTERFACE

type capability = { **mutable** cap_port: port ;
mutable cap_priv: privat }

type header = { **mutable** h_port: port ;
mutable h_priv: privat ;
mutable h_command: command ;
mutable h_status: status ;
mutable h_offset: int ; ▷ *32 bit* ◁
mutable h_size: int ; ▷ *16 bit* ◁
mutable h_extra: int } ▷ *16 bit* ◁

Basic Functions and Values

Functions to manage and manipulate ports, capabilities and headers are provided.

PROGRAMMING INTERFACE

[port] = **port_new** ()

[privat] = **priv_new** ()

[capability] = **cap_new** ()

[bool] = **port_cmp** Module: →
p2: port

[bool] = **nullport** port

[*port_value*: int] = **get_portbyte** ~port : port →
~byte : int

[unit] = **set_portbyte** ~port : port →
~byte : int

The *XX_new* functions return fresh values of the specific type. The *port_cmp* function test two ports for equality. The result is a boolean value. The *null_port* function tests for a zero port (all

bytes zero). The *XX_portbyte* functions are used to modify single bytes from a port value.
For each basic structure, there is a so called nil value - a dummy value for initial reference assignments, for example

```
let ref_port = ref nilport
```

PROGRAMMING INTERFACE

val nilport: port

val nilpriv: privat

val nilcap: capability

val nilheader: header

Some care must be taken in the case of multithreaded programming in *OCaML* (the default case in VAM). Because of the highly degree of optimisation in *OCaML*, different threads using the same function or module can share physically the same variables with undeterministic behaviour. To get a physical new copy of an existing value, there are some copy functions:

PROGRAMMING INTERFACE

[command] = **cmd_copy** command

[status] = **stat_copy** status

[hedare] = **header_copy** header

Encryption and Rights

Amoeba uses currently a standard 48 bit data encryption service. To encrypt a port value, internaly the *one_way* function is used. This function uses itself the *Des48* module for the real encryption. The user level function *priv2pub* is equipped with an additional cache for already encrypted ports. This function is used to convert a private server port to a public client port, needed for the Remote Procedure Call communication layer. The private ports are only used by the server *getreq* function, and the public port is used only by the client *trans* function. See the ????? module for details.
The *uniqport* function creates a new random port.

There are two functions for encoding and decoding of private parts from a capability:

- *prv_encode* → encode a private checkfield port with an object number and specific rights to a new private field for a capability
- *prv_decode* → decode a private part of a capability and verify the private checkfield port again

To extract the rights field and the object number from a private part of a capability, the *prv_rights* and *prv_number* functions are provided. Because the rights field in a capability can be manipulated by anyone, and for example a capability with restricted rights is manipulated to full rights, always the *prv_decode* function must be used to verify the correctness of the private field.

PROGRAMMING INTERFACE

val prv_all_rights: rights_bits

[oport] = **one_way** *iport*: port

[*pubport*: port] = **priv2pub** *prvport*: port

[bool] = **prv_decode** ~prv : privat →
~rand : port

[*en_privat*: privat] = **prv_encode** ~obj : int →
~rights : rights_bits →
~rand : port

[obj_num] = **prv_number** privat

[rights_bits] = **prv_rights** privat

[*newport*: port] = **uniqport** ()

There are several functions to manipulate and check rights fields. The *rights_req* functions expects a rights field and a required rights list. The *rights_set* function ca be used to create a rights field from a rights list.

PROGRAMMING INTERFACE

[rights_bits] = **rights_and _or _xor** rights_bits →
rights_bits

[rights_bits] = **rights_not** rights_bits

[bool] = **rights_req** ~rights : rights_bits →
~required : rights_bits list

[rights_bits] = **rights_set** rights_bits list

This module delivers the programmer with a collection of functions to convert Amoeba basic structures like ports and capabilies in an ASCII string representation and vice versa.

Amoeba to String

PROGRAMMING INTERFACE

[string] = **ar_port** *port*: port

[string] = **ar_priv** *private*: privat

[string] = **ar_cap** *cap*: capability

[string] = **ar_cs** *cs*: capset

These functions convert an Amoeba port, a private field and a capability into a string. The string output has the following format:

```
Port:          x:x:x:x:x
Private:      d(r)/y:y:y:y:y:y
          d:  object number
          (r):  rights field
Capability:     x:x:x:x:x:x/d(r)/y:y:y:y:y:y

d:  decimal value
x,y,r:  hexadecimal value
```

String to Amoeba

PROGRAMMING INTERFACE

[port] = **ar_toport** *sport*: string

[privat] = **ar_topriv** *sprivate*: string

[capability] = **ar_tocap** *scap*: string

These functions convert ports, private fields and capabilities from the string format shown above into values. The *ar_cap* and *ar_tocap* functions are also available under the names *c2a* and *a2c*.

Amoeba boot (kernel) directory partition read and write support routines.
The Amoeba boot partition is a very simple complete filesystem containing:

- Kernel images,
- Binaries for bootstrap purposes,
- configuration files and many other user customized things.

This partition, managed by the virtual disk server in a low elevel way, and usually labeled vdisk:01, is also read by the Amoeba boot manager.
The filesystem consists only of up to *bde_NENTRIES* (=21) files specified with a name string of maximal length *bde_NAMELEN* (=16).
There are function to store and retrieve the boot directory to and from generic bytebuffers.

PROGRAMMING INTERFACE *Types and Strcutures*

val bd_MAGIC: word32

val bd_NENTRIES: int

val bde_NAMELEN: int

type bootdir_entry = { **mutable** bde_start: word32 ; ▷ *start sector / block* ◁
mutable bde_size: word32 ; ▷ *size in sectors / blocks* ◁
mutable bde_name: string } ▷ *name of entry - bde_NAMELEN* ◁

type bootdir = { **mutable** bd_magic: word32 ;
mutable bd_entries: bootdir_entry array ;
mutable bd_unused: word32 }

PROGRAMMING INTERFACE *Functions*

[*pos*: int ∗
bootdir_entry] = **buf_get_bde** ~buf : buffer →
~pos : int

[*pos*: int ∗
bootdir] = **buf_get_bd** ~buf : buffer →
~pos : int

[*pos*: int] = **buf_put_bde** ~buf : buffer →
~pos : int →
~bdes : bootdir_entry

[*pos*: int] = **buf_put_bd** ~buf : buffer →
~pos : int →
~bd : bootdir

[string] = **bde_name** *bname*: string

Module dependencies

- Amoeba
- Bytebuf
- Buf
- Machtype

Generic byte stream support for stream like servers, for example Amoeba's serial port server residing in the kernel.

stream_read
read max. # size bytes from stream server specified with srv capability.

stream_buf_read
same as above, but read into a buffer starting at pos of maximal length size. Returns # of received bytes.

stream_write
write # size bytes to stream server specified with srv capability.

stream_buf_write
same as above, but write from a buffer starting at pos of maximal length size.

PROGRAMMING INTERFACE

```
[ status *
  string   ] = stream_read ~srv : capability →
                           ~size : int

[ status *
  n: int   ] = stream_buf_read ~srv : capability →
                               ~buf : buffer →
                               ~pos : int →
                               ~size : int

[ status ] = stream_write ~srv : capability →
                          ~str : string

[ status ] = stream_buf_write ~srv : capability →
                              ~buf : buffer →
                              ~pos : int →
                              ~size : int
```

Module dependencies

- Amoeba
- Bytebuf

These functions are used to store and extract Amoeba strcutures and values to and from Amobea buffers in a mechine independet way.

Buffer Put Functions

Integer values, 16 and 32 bit wide, are stored with the *but_put_int16* and the *buf_put_in32* functions. OCaML integers can be stored with the *buf_put_int* function. They expect the buffer *buf* , the start position *pos* within the buffer and the int value *int16* or *int32* or *int* as their arguments.
Strings can be stored with the *buf_put_string* function at the specified start position in the given buffer. Each string is closed with the null character *'\000'*.
Ports, private parts and capabilities and capability sets are stored with the *buf_put_port*, *buf_put_priv*, *buf_put_cap* and *buf_put_capset* functions.
Rights can be stored in two different ways using the *buf_put_right_bits* and *buf_put_rights_bits* functions.
All functions return the next position within the buffer, or raise the *Buf_overflow* exception.

PROGRAMMING INTERFACE

[*newpos*: int] = **buf_put_int16** ~buf : buffer →
~pos : int →
~int16 : int

[*newpos*: int] = **buf_put_int32** ~buf : buffer →
~pos : int →
~int32 : int

[*newpos*: int] = **buf_put_int** ~buf : buffer →
~pos : int →
~int : int

[*newpos*: int] = **buf_put_string** ~buf : buffer →
~pos : int →
~str : string

[*newpos*: int] = **buf_put_port** ~buf : buffer →
~pos : int →
~port : port

[*newpos*: int] = **buf_put_priv** ~buf : buffer →
~pos : int →
~priv : privat

[*newpos*: int] = **buf_put_cap** ~buf : buffer →
~pos : int →
~cap : capability

[*newpos*: int] = **buf_put_capset** ~buf : buffer →
~pos : int →
~cs : capset

[*newpos*: int] = **buf_put_right_bits** ~buf : buffer →
~pos : int →
~right : int

[*newpos*: int] = **buf_put_rights_bits** ~buf : buffer →
~pos : int →
~rights : rights_bits

Buffer Get Functions

Integer values, 16 and 32 bit wide, are extracted from buffers using the *but_get_int16* and the *buf_get_in32* functions. Ordinary OCaML integers can be extracted with the *buf_get_int* function. They expect the buffer *buf* and the start position *pos* within the buffer, and return the next buffer position and the int value *int16* or *int32* or *int*.
Strings can be extracted with the *buf_get_string* function at the specified start position in the given buffer. The functions expects a null character terminated string.
Ports, private parts and capabilities and capability sets are extracted with the *buf_get_port*, *buf_get_priv*, *buf_get_cap* and *buf_get_capset* functions.
Rights can be extracted in two different ways using the *buf_get_right_bits* and *buf_get_rights_bits* functions.
All functions return the next position within the buffer, or raise the *Buf_overflow* exception.

PROGRAMMING INTERFACE

[*newpos*: int ∗
int16: int] = **buf_get_int16** ~buf : buffer →
~pos : int

[*newpos*: int ∗
int32: int] = **buf_get_int32** ~buf : buffer →
~pos : int

[*newpos*: int ∗
int: int] = **buf_get_int** ~buf : buffer →
~pos : int

[*newpos*: int ∗
str: string] = **buf_get_string** ~buf : buffer →
~pos : int

[*newpos*: int ∗
port: port] = **buf_get_port** ~buf : buffer →
~pos : int

[*newpos*: int ∗
priv: privat] = **buf_get_priv** ~buf : buffer →
~pos : int

[*newpos*: int ∗
cap : capability] = **buf_get_cap** ~buf : buffer →
~pos : int

[*newpos*: int ∗
cs: capset] = **buf_get_capset** ~buf : buffer →
~pos : int

[*newpos*: int ∗
right: int] = **buf_get_right_bits** ~buf : buffer →
~pos : int

[*newpos*: int ∗
rights: rights_bits] = **buf_put_rights_bits** ~buf : buffer →
~pos : int

File utils

There are some functions to read and write Amoeba types in an operating system and machine independent way to and from files.

PROGRAMMING INTERFACE

[status ∗
cap] = **read_cap** *filename*: string

[status] = **write_cap** *filename*: string →
cap: capability

Module Dependencies

- ➤ *Amoeba*
- ➤ *Bytebuf*
- ➤ *Capset*
- ➤ *Os*
- ➤ *Int32*

Generic cache module. Provides a fixed table cache. The fixed table is treated like a circular buffer, therefore if filled up, the oldest entry is overwritten. Seraching is done from the newest entry downto the oldest.
Assumption: the cache is mostly filled up. It's possible to invalidate (remove) a cache entry.

PROGRAMMING INTERFACE

type ('a,'b) cache_entry = { cache_key : 'a ;
cache_data : 'b }

type ('a,'b) t = { **mutable** cache_size : int ;
mutable cache_head : int ;
mutable cache_hit : int ;
mutable cache_miss : int ;
mutable cache_table : (('a,'b) cache_entry) option array }

[('a,'b) t] = **create** ~size : int

[unit] = **add** ~cache : ('a, 'b) t →
~key : 'a →
~data : 'b

['b] = **lookup** ~cache : ('a, 'b) t →
~key : 'a

[unit] = **invalidate** ~cache : ('a, 'b) t →
~key : 'a

[Environment capabilities $\equiv$ local named capabilities]
Environment capabilities like the root directory capability or the tty server capability, can be extracted either in native Amoeba from the process environment, or under Unix from the user shell environment variables.
Suppose the case the VAM is running under Unix, and a VAM program want to know his directory root capability, the user must export the shell environment variable with the capability in the common ASCII representation (for example in the shell profile file):

```
ROOTCAP="a1:de:14:f:1d:cf/1(ff)/29:51:4b:0:ba:bb"
export ROOTCAP
or
ROOTCAP=/unix/amoeba/dns/.servercap
export ROOTCAP    # Get the capability from a UNIX file
TTYCAP=/server/tty
export TTYCAP    # Lookup the cap from the directory server
```

It's also possible to give these environment variables an absolute path name for a file, which holds the capability (stored with the *buf_put_cap* / *write_cap* function).
The path name must be preceeded by the '/unix' prefix to indicate a UNIX path, else the capability is lookuped from a directory server (DNS). Note: to avoid variable name conflicts the UNIX name of the environment variable ends with the CAP specifier. In VAM/Amoeba, this endings is recognized and elimintaed, and you can lookup the Amoeba environment variable name without this ending.
Within the VAM program, the function *get_env_cap* can be used to extract the capability. In addition to the get_env_cap function, there is a *put_env_cap* function to create or change environment variables - but only in the context of the current program. To perform this task, an environment name-capability hash is used.
Additionally, there are functions to resolve a path name to either an UNIX or an Amoeba path.

Programming Interface

[status $*$
capability] = **get_env_cap** *envname*: string

[status] = **put_env_cap** *envname*: string $\rightarrow$
cap: capability

type path_arg = Amoeba_path of string
| Unix_path of string

[path_arg] = **path_resolve** path

Amoeba capability set utilities.

cs_singleton
Convert a capability to a cap set.

cs_goodcap
Get a useable capability from a capset, and retrun this cap. Returns the first capability in the set for which std_info returns STD_OK. If there are no caps in the set for which std_info returns STD_OK, then the last cap in the set is returned and the err status STD_INFO.

cs_to_cap
Get a capability from a capset giving preference to a working capability. If there is only one cap in the set, this cap is returned. If and only if there is more than one, try std_info on each of them to obtain one that is useable, and return this one. If none of the multiple caps work, the last one is returned. Callers who need to know whether the cap is useable should use cs_goodcap(), above. Returns STD_OK, unless the capset has no caps, in which case, returns STD_SYSERR.

cs_copy
Return a fresh capset and copy the original contents.

Programming Interface

type suite = { **mutable** s_object : capability ;
mutable s_cuurent : bool }

type capset = { **mutable** cs_initial : int ;
mutable cs_final : int ;
mutable cs_suite : suite array }

val nilcapset: capset

[capset] = **cs_singleton** capability

[status ∗
capability] = **cs_goodcap** capset

[status ∗
capability] = **cs_to_cap** capset

[capset] = **cs_copy** capset

Circular buffer package.
Circular buffers are used to transfer a stream of bytes between a reader and a writer, usually in different threads. The stream is ended after the writer closes the stream; when the reader has read the last byte, the next read call returns an end indicator. Flow control is simple: the reader will block when no data is immediately available, and the writer will block when no buffer space is immediately available. This package directly supports concurrent access by multiple readers and/or writers.

cb_create
: Allocates a new circular buffer of given size.

cb_close
: Closes circular buffer and set closed flag. May be called as often as you want, by readers and writers. Once closed, no new data can be pushed into the buffer, but data already in it is still available to readers.

cb_full
: Returns number of available data bytes. ** When closed and there are no bytes available, return -1.

cb_empty
: Returns number of available free bytes. Return -1 if closed (this can be used as a test for closedness).

cb_putc
: Puts one char into the cicrcular buffer. Returns 1 if OK, else -1 if closed.

cb_puts
: Puts n chars into the circular buffer. Returns number of written chars, or -1 if cb is closed.

cb_getc
: Gets the next byte from the circular buffer and returns it in char converted form. Returns always true and CB content char if OK, else false,'\000' if closed and no more data is available.

cb_trygetc
: Tries to get a char from the circular buffer. Returns false if closed or no chars are available. May be interruptable.

cb_gets
: Gets between minlen and maxlen chars from circbuf. Returns a new string of length (minlen < avail_bytes < maxlen). Returns empty string if cb was closed.

cb_putb
: Puts one byte (integer) into the circular buffer. Returns 1 if OK, else -1 if closed.

cb_putbn
: Puts n bytes, stored in the generic buffer area, into the cb. Returns number of written bytes, or -1 if cb is closed.

cb_getb
: Get next byte from the cb and returns it in integer converted form. Returns always true and CB byte content if OK, else false if closed and no more data is available.

cb_trygetb
: Tries to get one byte.

cb_getbn
: Gets between minlen and maxlen bytes from circbuf and stores the data in the given generic buffer. Returns -1 if cb closed.

cb_getsn
Gets between minlen and maxlen bytes from circbuf and stores the data in the specified string area. Returns -1 if cb closed. String version.

cb_putsn
Puts n bytes, stored in the given string arera, into the cb. Return number of written bytes, or -1 if cb is closed. String version.

cb_getp, cb_getpdone
Gets the position for the next output byte in the buffer. Returns (-1,-1) if cb closed, (0,-1) if no bytes available, else (num,pos) of available bytes, but limited to the upper bound of the cb, and the position within the buffer. If nonzero return, a call to cb_getpdone must follow to announce how many bytes were actually consumed.

cb_putp, cb_putpdone
Gets the position for the next free byte in the buffer. Returns (-1,-1) if cb closed, (0,-1) if no free bytes available, else (num,pos) of available bytes, but limited to the upper bound of the cb, and the position within the buffer. If nonzero return, a call to cb_putpdone must follow to announce how many bytes were actually stored.

PROGRAMMING INTERFACE

[circular_buf] = **cb_create** ~size : int

[unit] = **cb_close** circular_buf

[*n*: int] = **cb_full** circular_buf

[*n*: int] = **cb_empty** circular_buf

[int] = **cb_putc** ~circbuf : circular_buf →
~chr : char

[int] = **cb_puts** ~circbuf : circular_buf →
~str : string

[bool *
char] = **cb_getc** circular_buf

[bool *
char] = **cb_trygetc** circular_buf

[string] = **cb_gets** ~circbuf : circular_buf →
~minlen : int →
~maxlen : int

[int] = **cb_getsn** ~circbuf : circular_buf →
~dst : string →
~dstpos : int →
~minlen : int →
~maxlen : int

```
[ int ] = cb_putsn ~circbuf : circular_buf →
                   ~src : string →
                   ~srcpos : int →
                   ~len : int

[ int ] = cb_putb ~circbuf : circular_buf →
                  ~byte : int

[ int ] = cb_putbn ~circbuf : circular_buf →
                   ~dst : buffer →
                   ~dstpos : int →
                   ~len : int

[ bool *
  int    ] = cb_getb circular_buf

[ int ] = cb_getbn ~circbuf : circular_buf →
                   ~src : buffer →
                   ~srcpos : int →
                   ~minlen : int →
                   ~maxlen : int

[ num: int *
  pos: int   ] = cb_getp circular_buf

[ unit ] = cb_getpdone circular_buf →
                       int

[ num: int *
  pos: int   ] = cb_putp circular_buf

[ unit ] = cb_putpdone circular_buf →
                       int
```

This file contains the list of first and last command codes assigned to each registered server. Only registered servers are listed. If you wish to register a new servers then email to sci@bsslab.de with your request for registration. The set of error codes is the negative of the command codes. Note that the RPC error codes are in the range RESERVED_FIRST to RESERVED LAST.
Registered commands take numbers in the range 1000 to (NON_REGISTERED_FIRST - 1).
Developers may use command numbers in the range NON_REGISTERED_FIRST to (NON_REGISTERED_LAST - 1) .
You should make all your command numbers relative to these constants in case they change in Amoeba 4.
Each server is assigned commands in units of 100. If necessary a server may take more two or more consecutive quanta. Command numbers 1 to 999 are reserved and may NOT be used.
The error codes that correspond to these command numbers are for RPC errors. Command numbers from 1000 to 1999 are reserved for standard commands that all servers should implement where relevant.

Double linked circular list implementation.

ML implementation of the basic algortihms needed for port encryption.

High level directory service stubs.

dir_lookup
Returns the (status,capability) tuple for the directory lookup of name. The server capability is an optional argument.

dir_append
Append a new object capability under with the given name to the directory tree.

dir_rename
Rename a directory entry.

dir_set_colmasks
This function sets the default column masks used when appending names with the dir/name interface. It should be called when the masks as specified by the environment variable SPMASK are not what we need.

dir_delete
Delete a directory. Note: due to the fact that under Amoeba the directory and filesystemis independent, only the directory table and the strucure is deleted, not the content!

dir_create
Create a new directory for server 'server'. Returns the new directory capability set. This capset can then appended somewhere in the directory tree.

dir_open
UNIX like directory handling: "open" a directory and return directory descriptor.

dir_next
Get the next directory entry from the given directory descriptor.

dir_close
Close the previously opend directory.

Programming Interface

[status ∗
capability] = **dir_lookup** ~root : capability →
~name : string

[status] = **dir_append** ~root : capability →
~name : string →
~obj : capability

[status] = **dir_rename** ~dir : capability →
~oldname : string →
~newname : string

[unit] = **dir_set_colmasks** *cols*: int array →
len: int

[status] = **dir_delete** ~root : capability →
~name : string

[status ∗
newdir: capability] = **dir_create** ~server : capability

type dir_row = { **mutable** dr_name: string ;
mutable dr_time: int ;
mutable dr_cols: int array }

type dir_desc = { **mutable** dir_rows: dir_row array ;
mutable dir_curpos: int ;
mutable dir_ncols: int ;
mutable dir_nrows: int ;
mutable dir_colnames: string array }

[status ∗
dir_desc] = **dir_open** ~dir : capability

[dir_row] = **dir_next** ~dirdesc : dir_desc

[unit] = **dir_close** ~dirdesc : dir_desc

Virtual disk server interface. The virtual disk server - either within the kernel or outside, manages physical and logical disks (aka. partitions) together with Amoeba subpartitions (aka. vdisks).

disk_info
This is the client stub for the disk_info command. It returns a list of disk_addr's which are tuples of (unit, firstblock, # blocks).

disk_read
This is the client stub for the disk_read command. Since reads may be bigger than fit in a single transaction we loop doing transactions until we are finished. If it can read exactly what was requested it succeeds. Otherwise it fails. No partial reads are done.

disk_write
This is the client stub for the disk_write command. Since writes may be bigger than fit in a single transaction we loop doing transactions until we are finished.

PROGRAMMING INTERFACE

```
type disk_info = { disk_unit: int32 ;
                   disk_firstblk: int32 ;
                   disk_numblks: int32 }

[ status *
  disk_info ] = disk_info srv: capability

[ status ] = disk_read srv: capability →
                       ~start : int →        ▷ first block ◁
                       ~num : int →          ▷ number of blocks ◁
                       ~blksize : int →      ▷ block size in bytes ◁
                       ~buf : buffer →       ▷ buffer to write to ◁
                       ~pos : int            ▷ start positionin buf ◁

[ status ] = disk_write srv: capability →
                        ~start : int →
                        ~num : int →
                        ~blksize : int →
                        ~buf : buffer →      ▷ buffer to read from ◁
                        ~pos : int
```

Module dependencies

- Amoeba
- Bytebuf

- Rpc
- Stderr
- Stdcom
- Buf
- Machtype

Kernel and network trace client interface. Limited to maximal number of 32000 events.

This module enables usage of machine specific data types with fixed bit size in a machine independent way, similar to OCaML's Int32 and Int64 modules. Supported types:

Int8
Signed 8 bit integer type.

Int16
Signed 16 bit integer type.

Int32
Signed 32 bit integer type.

Int64
Signed 64 bit integer type.

Uint8,Word8
Unsigned 8 bit integer type.

Uint16,Word16
Unsingned 16 bit integer type.

Uint32,Word32
Unsigned 32 bit integer type.

Uin64,Word64
Unsigned 64 bit integer type.

All native OCaML artithemtic and logic operators are supported with integer, float and the machine type:

Arithmetic operators
+, -, ∗, /

Logic operators
land, lor,lsl, lsr

PROGRAMMING INTERFACE *Types*

```
type int8 = <abstr>

type int16 = <abstr>

type int32 = <abstr>

type int64 = <abstr>

type uint8 = <abstr>

type uint16 = <abstr>
```

type uint32 = <abstr>

type uint64 = <abstr>

type word8 = <abstr>

type word16 = <abstr>

type word32 = <abstr>

type word64 = <abstr>

type machtype_id = Int8
| Int16
| Int32
| Int64
| Uint8
| Uint16
| Uint32
| Uint64
| Word8
| Word16
| Word32
| Word64

PROGRAMMING INTERFACE *Type conversion*

[int] = **to_int** 'a

['a] = **of_int** int →
machtype_id

[string] = **to_str** 'a

['a] = **of_str** string →
machtype_id

[string] = **format** string →
'a

[string] = **to_data** 'a

['a] = **of_data** string →
machtype_id

[int8] = **int8** int

[int16] = **int16** int

[int32] = **int32** int

[int8] = **int8s** string

[int16] = **int16s** string

[int32] = **int32s** string

[uint8] = **uint8** int

[uint16] = **uint16** int

[uint32] = **uint32** int

[uint8] = **uint8s** string

[uint16] = **uint16s** string

[uint32] = **uint32s** string

[word8] = **word8** int

[word16] = **word16** int

[word32] = **word32** int

[word8] = **word8s** string

[word16] = **word16s** string

[word32] = **word32s** string

[int] = **int** 'a

PROGRAMMING INTERFACE *Buffer management*

[*pos*: int] = **buf_put_mach** ~buf : buffer →
~pos : int →
~mach : 'a

[*pos*: int ∗
'a] = **buf_get_mach** ~buf : buffer →
~pos : int →
~mach : machtype_id

Server Event Monitoring Module.
All event strings are referenced by a circular cache with the 'event' function. A client can extract all events from the cache with the 'event_get' function. An event can only be read one time.
Each module or the whole program can start a separate server thread with the 'event_server' function together with the event structure, previously created with the 'event_init' function (== 'event_start').
The 'event_init' functions expect the number of cache entries and the portname string, which is converted in a private port. This port name must also be given to the 'event_get' function.

Directory tree name services. This module provides a simple way for server to publish theit server capabilities in the Amoeba directory system and for clients to lookup these capabilities simply providing the pathname.

Programming Interface

[status $*$
capability] = **name_lookup** *path*: string

[status] = **name_append** ~name : string $\rightarrow$
~cap : capability

[status] = **name_delete** *path*: string

Low level process execution module. This is the most complicated interface in this system.

Programmers API for Amoeba Remote Procedure Calls, building the core concepts for distributed programming and environments.

getreq
: The server side. The server starts requetsing on the specified server port (private port). After a client send a transaction, this function returns with the request header and the actual amount of data stored in the request buffer. Because the RPC is symmetrical, a putrp must follow this function!

putrep
: Send a reply upon a client rqeuest.

trans
: Client side. Send a transaction to the server specified in the request header. The data buffer contains request data or can be replaced with the nilbuf and request size = 0. After the server responded, teh reply buffer is filled with the reply data (if any) and the reply transaction header is retruned together with the number of byted received in the reply buffer.

timeout
: Specify the maximal time to lookup/search for a server. If a transaction can't be trasnferred in this time, the status of this operation is set to *RPC_NOTFOUND*.

The XXXo functions support buffer offset specifiers (request and reply buffers).

PROGRAMMING INTERFACE

[*stat*: status ∗
repsize: int ∗
hdrrep: header] = **trans** (hdrreq:header ∗ bufreq:buffer ∗ reqsize:int ∗ bufrep:buffer ∗ bufsize:int)

[*stat*: status ∗
reqsize: int ∗
hdrreq: header] = **getreq** (portreq:port ∗ bufreq:buffer ∗ bufsize:int)

[*stat*: status] = **putrep** (hdrrep:header ∗ bufrep:buffer ∗ repsize:int)

[*stat*: status ∗
repsize: int ∗
hdrrep: header] = **transo** (req:header ∗ buffer ∗ size:int ∗ off:int ∗ rep:buffer ∗ size:int ∗ off:int)

[*stat*: status ∗
reqsize: int ∗
hdrreq: header] = **getreqo** (portreq:port ∗ bufreq:buffer ∗ bufsize:int ∗ bufoff:int)

[*stat*: status] = **putrepo** (hdrrep:header ∗ bufrep:buffer ∗ repsize:int ∗ bufoff:int)

[status] = **timeout** *interval*: int ▷ *interval in milli second units* ◁

Example

The following code shows a simple RPC example.

```
open Amoeba
open Rpc
open Cmdreg
open Stdcom
open Stderr
open Bytebuf
open Buf


(*
** Standard restrict request.  Returns the restricted capability
** of the object 'cap' owned by the server.
*)

let std_restrict ~cap ~mask =
    let bufsize = cap_SIZE in
    let buf = buf_create bufsize in


    if (cap.cap_priv.prv_rights = prv_all_rights)then
    begin
        let obj = prv_number cap.cap_priv in
        let cap' = {
                    cap_port = cap.cap_port;
                    cap_priv =
                        prv_encode ~obj:obj
                                ~rights:mask
                                ~rand:cap.cap_priv.prv_random;
                } in
        std_OK,cap'

    end
    else
    begin
        let Rights_bits mask = mask in
        let hdr_req = {
                h_port = cap.cap_port;
                h_priv = cap.cap_priv;
                h_command = std_RESTRICT;
                h_status = std_OK;
                h_offset = mask;
                h_size = bufsize;
                h_extra = 0;
                } in

        let (err_stat,size,hdr_rep) = trans
                                (hdr_req,nilbuf,0,buf,bufsize)
            in

        if(size > 0 && (hdr_rep.h_status = std_OK)) then
```

```
        begin
            let pos,cap_restr = buf_get_cap ~buf:buf ~pos:0 in
            (err_stat,cap_restr)
        end
        else if (err_stat <> std_OK) then
            (err_stat,nilcap)
        else
            (hdr_rep.h_status,nilcap)
    end
```

In this example, the server port and the private field are taken from the given capability. After the *trans* function returns, first the status of the RPC operation must be checked. If *err=std_OK*, the server response status returned in the reply header must be checked.

Module dependencies

- Amoeba
- Bytebuf

Amoeba's standard operation requests.

std_info
Standard information rqeuest. Returns the server information string. The server capability is specified with 'cap'.

std_status
Standard status request. Returns the server status string (statistical informations). The server capability is specified with 'cap'.

std_exit
Standard exit request. Send the server the exit command. The server capability is specified with 'cap'.

std_destroy
Destroy a server object, for example a memory segment.

std_touch
Touch a server object. This is a NOP, but increments the live time of an object, ify any.

std_age
Age all objects of a server (decrements the live time of all objects and destroys objects with live time equal to zero). Only allowed with the servers super capability and prv_all_rights!

PROGRAMMING INTERFACE

[status ∗
string] = **std_info** ~cap : capability →
~bufsize : int

[status ∗
string] = **std_status** ~cap : capability →
~bufsize : int

[status] = **std_exit** ~cap : capability

[status] = **std_destroy** ~cap : capability

[status] = **std_touch** ~cap : capability

[status] = **std_age** ~cap : capability

Some more standard requests.

std_restrict
Standard restrict request. Returns the restricted capability of the object 'cap' owned by the server.

std_exec
Standard exec request. Execute a string list on a server, for example MLor Forth scripts. The reply is returned in a string, too.

std_set_params
Set parameters for server administration. Format of argument list: <name>,<value>

std_get_params
Get parameter list from specified server. Returns string tuple format: <name>,<range and unit>,<desc>,<value>

Programming Interface

[status *
rcap: capability] = **std_restrict** ~cap : capability →
~mask : rights_bits

[status *
string] = **std_exec** ~srv : capability →
~args : string list

[status] = **std_set_params** ~srv : capability →
~args : (string * string) list

[status *
(string*string*string*string) list] = **std_get_params** ~srv : capability

Contains defintions for Amoeba standard error codes and a descriptive name list mapping error numbers with strings.

PROGRAMMING INTERFACE

[string] = **err_why** status

This file contains the definitions of the symbols used by servers in their stdinfo string to identify the object. Note: only objects are identified this way. Servers for the object describe themselves with a longer string at present, although they could be of the object type server and return S followed by the symbol of the type of object.

```
let objsym_TTY     = "+"          (* TTY                    *)
let objsym_BULLET  = "-"          (* Bullet File            *)
let objsym_AFS     = objsym_BULLET (* AFS File              *)
let objsym_DIRECT  = "/"          (* Directory              *)
let objsym_DNS     = objsym_DIRECT (* Directory             *)
let objsym_KERNEL  = "%"          (* Kernel Directory       *)
let objsym_DISK    = "@"          (* Disk, Virtual or Physical *)
let objsym_PROCESS = "!"          (* Process, Running or not *)
let objsym_PIPE    = "|"          (* Pipe ?                 *)
let objsym_RANDOM  = "?"          (* Random Number Generator *)
```

This module provides a simple implementation of Amoeba leightweighted signal concepts. Currently only the *sig_TRANS* and the *sig_INT* signals are supported. The first is used ins erver to catch client RPC interrupt signals, the second is used to catch user interrupt signals from the UNIX shell (CTRL-C).
The *sig_catch* function installs a signal handler thread of the specified signal for only one thread. The function argument holds the signal number.

Programming Interface

val sig_TRANS: int

val sig_INT: int

[unit] = **sig_catch** ~signum : int →
~handler : (int→unit)

Virtual circuit module, comparable with named pipes under UNIX. Implemented with two circular buffers and a client and server loop thread.

vc_create
Create a new virtual circuit. Full duplex capable. This function starts both the vc_client and the vc_server thread.

vc_close
Close one or both circular buffers.

vc_reads
Read a string with given maximal length from the virtual circuit, but at least minlen characters.

vc_readb
Reads instead in a generic buffer at position pos (maximal length len).

vc_writes
Write a string into the virtual circuit ring.

vc_writeb
Write to the vc from a buffer starting at position pos and length len.

vc_getp, vc_getpdone
Gets a circular buffer pointer to fetch data from = cb_getp vc.vc_cb.(client)

vc_putp, vc_putpdone
Gets a circular buffer pointer to store data in = cb_putp vc.vc_cb.(server)

PROGRAMMING INTERFACE

[virt_circ] = **vc_create** ~iport : port →
~oport : port →
~isize : int →
~osize : int

[unit] = **vc_close** virt_circ →
which: int

[int] = **vc_reads** virt_circ →
~str : string →
~pos : int →
~len : int

[int] = **vc_readb** virt_circ →
~buf : buffer →
~pos : int →
~len : int

[int] = **vc_writes** virt_circ →

~str : string →
~pos : int →
~len : int

[int] = **vc_writeb** virt_circ →
~buf : buffer →
~pos : int →
~len : int

[*num*: int ∗
pos: int] = **vc_getp** virt_circ

[unit] = **vc_getpdone** virt_circ →
int

[*num*: int ∗
pos: int] = **vc_putp** virt_circ →
int

[unit] = **vc_putpdone** virt_circ →
int

Low level Buffer management.

Basic functions

The *buf_physical* function creates a new master buffer of the specified size. Physical memory space will be allocated by this function. The *buf_logical* function derives a slave buffer from a master buffer. The slave buffer is a window from the master buffer. No additional data memory space will be allocted by this function. The *buf_copy* creates a new physical buffer and copies the content of the source buffer *src* starting at position *pos* of size *size* to the new one.
There are several get and set functions to extract or store values into a buffer at a specific position.

PROGRAMMING INTERFACE

exception *Buf_overflow*

external buf_physical: size : int →
buffer
= **"ext_buf_physical"**

external buf_logical: *src*: buffer →
pos: int →
size: int →
buffer
= **"ext_buf_logical"**

external buf_copy: *src*: buffer →
pos: int →
size: int →
buffer
= **"ext_buf_copy"**

external buf_get: buffer → ▷ *buf* ◁
int → ▷ *pos* ◁
int
= **"ext_buf_get"**

external buf_set: buffer → ▷ *buf* ◁
int → ▷ *pos* ◁
int → ▷ *byte* ◁
()
= **"ext_buf_set"**

external buf_gets: buffer → ▷ *buf* ◁
int → ▷ *pos* ◁
int → ▷ *len* ◁
string
= **"ext_buf_gets"**

external buf_sets: buffer → ▷ *buf* ◁
int → ▷ *pos* ◁
string → ▷ *str* ◁
()
= **"ext_buf_sets"**

external buf_getc: buffer → ▷ *buf* ◁
int → ▷ *pos* ◁
char
= **"ext_buf_getc"**

external buf_setc: buffer → ▷ *buf* ◁
int → ▷ *pos* ◁
char → ▷ *char* ◁
()
= **"ext_buf_setc"**

external buf_len: buffer →
int
= **"ext_buf_len"**

String module compatibility

To make compatibility the *Bytebuf* compatible with the *String* module, there are several functions to convert strings to buffers and vice versa, to get and set values in a buffer, and various blit functions known from the string module.

PROGRAMMING INTERFACE

[string] = **buf_tostring** *buf*: buffer →
pos: int →
len: int

[buffer] = **buf_ofstring** *str*: string →
pos: int →
len: int

val nilbuf: buffer

[int] = **length** buffer

[char] = **get** *buf*: buffer →
pos: int

[unit] = **set** *buf*: buffer →
pos: int →

c: char

[buffer] = **create** *size*: int

[buffer] = **copy** buffer

[buffer] = **sub** *buf*: buffer →
~pos : int →
~len : int

[string] = **string_of_buf** buffer

[buffer] = **buf_of_string** string

external blit_bb: *src*: buffer →
src_pos: int →
dst: buffer →
dst_pos: int →
len: int →
unit
= **"ext_blit_bb"**

external blit_bs: *src*: buffer →
src_pos: int →
dst: string →
dst_pos: int →
len: int →
unit
= **"ext_blit_bb"**

external blit_sb: *src*: string →
src_pos: int →
dst: buffer →
dst_pos: int →
len: int →
unit
= **"ext_blit_bb"**

external fill: buffer →
pos: int →
len: int →
int →
unit
= **"ext_fill"**

external buf_info: buffer →
string
= **"ext_buf_info"**

Device Driver Interface for Amoeba. Only implemented in the vamrun version running on the top of a native Amoeba kernel (AMOEBA_RAW).

IO Port Management

To access IO ports, the IO port address must be registered by the kernel. Not registered IO access causes a memory violation error.

io_check_region
Check an IO port region, starting at address 'start' and with an extent of 'size' bytes. If status std_Ok was returned, this IO region can be mapped in the current process. The system capability is currently the root capability of the kernel.

io_map_region
Map in an already checked IO port region. After this (successfull) call, IO ports can be read and written using the funtions below. Access to IO ports not mapped in the process address space raises a memory access violation exception. The system capability is currently the root capability of the kernel.

io_unmap_region
Unmap a previously mapped IO port region, starting at address 'start' and with an extent of 'size' bytes.

PROGRAMMING INTERFACE

```
[ status ] = io_check_region ~start : int32 →
                             ~size : int32 →
                             ~syscap : capability

[ status ] = io_map_region ~start : int32 →
                           ~size : int32 →
                           ~devname : string →
                           ~syscap : capability

[ status ] = io_unmap_region ~start : int32 →
                             ~size : int32 →
                             ~syscap : capability
```

IO Port Access

To access IO ports, the IO port address must be registered by the kernel. Not registered IO access causes a memory violation error.

out_byte
Write a byte value to an IO port. The port address must be mapped in the process.

in_byte
Read a byte value from an IO port. The port address must be mapped in the process.

PROGRAMMING INTERFACE

[unit] = **out_byte** ~addr : int32 →
~data : int32

[int32] = **in_byte** ~addr : int32

Timer

Software timers.

timer_init
The timer_init function initializes and installs a new software interval timer. The user specified event 'ev' will be wakedup after the time interval period in unit (SEC, MILLISEC, MICROSEC) relative to the current system time has elapsed. If the once argument is equal zero, the timer function will be called periodically, else only one time.

timer_reinit
Same as above, but timer settings of an already installed timer handler can be modified. If the specified timeout value is zero, the timer handler is removed. Note: before a process exits, it must currently remove the installed timer handler before exiting!

timer_await
Wait for the timer event Returns negative value if the call was interrupted.

timer_create_event
Create a timer event.

PROGRAMMING INTERFACE

type timer_event = Thread.thread_event

[timer_event] = **timer_create_event** ()

[int] = **timer_init** ~event : timer_event →
~interval : int →
~uni : time_unit →
~once : bool

[int] = **timer_reinit** ~event : timer_event →
~interval : int →
~uni : time_unit →
~once : bool

[int] = **timer_await** timer_event

Module dependencies

- Amoeba
- Thread
- Machtype

This package contains modules for building the *AFS* and *DNS* servers. The *AFS* server provides the Atomic Filesystem service for the Amoeba system, and the *DNS* server provides a generic object naming and capability mapping service with tree structures, aka. the directory server. Additionally it provides modules of the client interfaces for these services.

This module contains values common to the client and server interface.

AFS requests

PROGRAMMING INTERFACE

val afs_CREATE: command ▷ *create a file* ◁

val afs_DELETE: command ▷ *delete a part of a file* ◁

val afs_FSCK: command ▷ *check filesystem* ◁

val afs_INSERT: command ▷ *insert data in an unlocked file* ◁

val afs_MODIFY: command ▷ *modify data of an unlocked file* ◁

val afs_READ: command ▷ *read data from a file* ◁

val afs_SIZE: command ▷ *get the file size* ◁

val afs_DISK_COMPACT: command ▷ *compact the disk* ◁

val afs_SYNC: command ▷ *flush all caches* ◁

val afs_DESTROY: command ▷ *destroy a file* ◁

val afs_REQBUFSZ: int ▷ *size of server request buffer* ◁

Rights

PROGRAMMING INTERFACE

val afs_RGT_CREATE: rights_bits

val afs_RGT_READ: rights_bits

val afs_RGT_MODIFY: rights_bits

val afs_RGT_DESTROY: rights_bits

val afs_RGT_ADMIN: rights_bits

val afs_RGT_ALL: rights_bits

Commit flags

PROGRAMMING INTERFACE

val afs_UNCOMMIT: int

val afs_COMMIT: int

val afs_SAFETY: int

Module Dependencies

- *Amoeba*
- *Cmdreg*
- *Stderr*
- *Stdcom*

Client user interface for the Atomic Filesystem Server *AFS* .

File requests

The following table shows the client file requests and the required rights for these operations. The meaning of the function arguments are explained.
First, a file object is created with the *afs_create* function. The new capability is returned. If the file is still unlocked (see below), the file can be modificated with the functions *afs_modify*, *afs_insert* and *afs_delete*.
The *commit* flag can have the following values:

afs_UNCOMMIT
: Don't commit (= lock) the file after the current operation. Further modifications of the file are still possible untill the file will be locked with a following request.

afs_COMMIT
: Commit (=lock) the file after this operation. No further modifications of the file data are possible. But still pending cache flushes are not performed.

afs_SAFETY
: Same as above, but the file data is written through the cache. The file is sync'ed with the disk.

A modification request on a locked file will result in a physical copy with a new capability returned and the desired modifications.
It's possible to write a file with one request *afs_CREATE*, but usually several *afs_MODIFY* request are used to write the file in medium sized fragments.
An *afs_MODIFY* request with *size=0* can be used to lock a file only.

(TAB. 17)

Request	Function arguments	Required Rights
afs_CREATE Create a new file object	*cap* capability of an already existing file or super cap *commit* Commit flag *buf* the data buffer *size* initial size	*afs_RGT_CREATE*
afs_MODIFY Modify the content of an unlocked file (overwrite or/and append)	*cap* capability of the file object *offset* file offset from where to modify data *size* size of modified data	*afs_RGT_MODIFY*
afs_INSERT Insert a part into the content of an unlocked file	see above	*afs_RGT_MODIFY*
afs_DELETE Delete a part of the content of an unlocked file	see above	*afs_RGT_MODIFY*
afs_READ Read the content from a file object	*offset* file offset where to start reading data *buf* buffer to read data in *size* data size to be read	*afs_RGT_READ*
afs_DESTROY Destroy a file object	*cap* the capability of the file object to be destroyed	*afs_RGT_DESTROY*
afs_SIZE Get the size of a file object	-	*afs_RGT_READ*

PROGRAMMING INTERFACE

```
[ err: status *
  size: int        ] = afs_size  ~cap : capability

[ err: status *
  newfile: capability  ] = afs_delete  ~cap : capability →
                                       ~offset : int →
                                       ~size : int →
                                       ~commit : int

[ err: status *
  newcap: capability  ] = afs_create  ~cap : capability →
                                      ~buf : buffer →
                                      ~size : int →
                                      ~commit : int

[ err: status *
  bytes: int       ] = afs_read  ~cap : capability →
                                 ~offset : int →
                                 ~buf : buffer →
```

~size : int

[*err*: status ∗
newcap: capability] = **afs_modify** ~cap : capability →
~buf : buffer →
~size : int →
~offset : int →
~commit : int

[*err*: status ∗
newcap: capability] = **afs_insert** ~cap : capability →
~buf : buffer →
~size : int →
~offset : int →
~commit : int

[*err*: status] = **afs_destroy** ~cap : capability

The following example shows the creation of a file:

```
let b = buf_create 50000 in
...
    let stat,cap' = afs_create ~cap:supercap
                        ~buf:b
                        ~size:5000
                        ~commit:afs_UNCOMMIT in
    if (stat <> std_OK) then
        failwith "AFS create failed";
...
    let stat',cap'' = afs_modify ~cap:cap'
                          ~buf:b ~size:45000
                          ~offset:5000
                          ~commit:afs_SAFETY in
    if (stat' <> std_OK) then
        failwith "AFS modify failed";
...
```

Administration requests

There are some administration requests to control caches, the file system and the server itself. The following table shows the available requests.

(TAB. 18)		
Request	**Function arguments**	**Required Rights**
afs_SYNC Flush all file caches	*server* capability of an already existing file or the super capability	*afs_RGT_READ*
afs_DISK_COMPACT Compact the file system. Remove fragmented wholes.	*server* super capability	*afs_RGT_ADMIN*
afs_FSCK Check the file system integredity.	see above	*afs_RGT_ADMIN*
std_EXIT Shutdown the server.	see above	*afs_RGT_ADMIN*

PROGRAMMING INTERFACE

[*err*: status] = **afs_sync** ~server : capability

[*err*: status] = **afs_fsck** ~server : capability

[*err*: status] = **afs_disk_compact** ~server : capability

Data structures and types

A file object entry bound by the file server inode table must have one of states given by the type *afs_file_state*. Each file owns an inode descriptor structure *afs_inode* and the file structure *afs_file* with all necessary informations about the file.
The main AFS structure: *afs_super*. This is the all known super structure with basic informations about the file system. This structure is generated by the server with fixed informations from the super block of the filesystem (name, nfiles, ports, size), and dynamically from the inode table (nfiles, nused, freeobj, nextfree).
The server structure *afs_server* must be filled by the main (user supplied) server module. It references server function supplied in higher server layers.
The server is responsible for managing the file table, for example caching, reading and writing of file changes. This is not part of this module!

afs_read_file
Read a file specified with his objnum (index) number. Physical reads only if not cached already (and a cache was implemented).

afs_modify_file
Modify data of a file. In the case, the (offset+size) fragment exceeds the current filesize, the filesize must be increased with *afs_modify_size* first.

afs_modify_size
Modify the size of the file object.

afs_commit_file
Commit a file to the disk. The flag argument specifies the way: immediately (*afs_SAFETY*) or later (*afs_COMMIT*) by the cache module if any.

afs_read_inode
Read an inode structure from disk (higher layer implementation).

afs_create_inode
Create a new inode with initial the *afs_file* structure. A true final flag indicates that the file size is final and not initial (*afs_file* with *afs_COMMIT*/*SAFETY* flag set).

afs_delete_inode
Delete an inode (file); free used disk space, if any.

afs_modify_inode
Modify an inode, for example the ff_live field was changed.

afs_read_super, afs_sync, afs_stat
Read the super structure, flush the caches (if any), create statistics informations.

afs_age, afs_touch
File object garbage collection: All file object known by the file server must be aged from time to time. All files reaching the live time 0 must be destroyed. But only file object nevermore used should be destroyed. Therefore, all file object still in used must be touched. This operation sets the live time for theses files to the maximal live time.

afs_exit
This function is called on server exit and must perform cleanups, if any.

PROGRAMMING INTERFACE

val afs_MAXLIVE: int ▷ *maximal obecjt livetime* ◁

type afs_file_state = FF_invalid ▷ *inode currently not used* ◁
| FF_unlocked ▷ *file currently unlocked* ◁
| FF_locked ▷ *file committed and written to disk |afs_SAFETY|* ◁
| FF_committed ▷ *file committed, but not synced |afs_COMMIT|* ◁

type afs_inode = { **mutable** fi_daddr:int ; ▷ *File disk address [blocks]* ◁
mutable fi_ioff: int ; ▷ *File inode logical offset [bytes]* ◁
mutable fi_res: int } ▷ *Reserved disk space for unlocked files [bytes]* ◁

type afs_file = { **mutable** ff_lock: Mutex.t ; ▷ *File lock* ◁
mutable ff_objnum: int ; ▷ *The directory index number* ◁
mutable ff_random: port ; ▷ *Random check number* ◁
mutable ff_time: int ; ▷ *Time stamp* ◁
mutable ff_live: int ; ▷ *Live time [0..MAXLIVETIME]* ◁
mutable ff_state: afs_file_state ;
mutable ff_size: int ; ▷ *Size of the file [bytes]* ◁
mutable ff_inode: afs_inode ; ▷ *Inode from higher layers* ◁
mutable ff_modified: bool } ▷ *modified afs_file?* ◁

type afs_super = { **mutable** afs_lock: Mutex.t ; ▷ *afs_super lock* ◁
mutable afs_name: string ; ▷ *filesystem label* ◁
mutable afs_nfiles: int ; ▷ *Number of total inode entries = files* ◁
mutable afs_nused: int ; ▷ *Number of currently used files* ◁
mutable afs_freeobjnums: int ; ▷ *Free slots list* ◁
mutable afs_nextfree: int ; ▷ *Next free slot* ◁
mutable afs_getport: port ; ▷ *Private server port (supercap)* ◁
mutable afs_putport: port ; ▷ *Public server port* ◁
mutable afs_checkfield: port ; ▷ *Random checkfield* ◁
mutable afs_block_size: int ; ▷ *Data blocksize [bytes]* ◁
mutable afs_nblocks: int } ▷ *Number of total data blocks* ◁

type afs_server = { **mutable** afs_super: afs_super ; ▷ *the super block* ◁
mutable afs_read_file: fun ; ▷ *Read a file* ◁
mutable afs_modify_file: fun ; ▷ *Modify a file* ◁
mutable afs_modify_size: fun ; ▷ *Change the file size* ◁
mutable afs_commit_file: fun ; ▷ *Commit the file* ◁
mutable afs_read_inode: fun ; ▷ *Read an inode of a file* ◁
mutable afs_create_inode: fun ; ▷ *Create a new inode for a file* ◁
mutable afs_delete_inode: fun ; ▷ *Delete an inode; free disk space* ◁
mutable afs_modify_inode: fun ; ▷ *Modified inode* ◁
mutable afs_read_super: fun ; ▷ *Read the super structure* ◁
mutable afs_sync: fun ; ▷ *Flush all caches* ◁
mutable afs_stat: fun ; ▷ *Get server and filesystem statistics* ◁
mutable afs_age: fun ; ▷ *Age a file* ◁
mutable afs_touch: fun ; ▷ *Touch a file* ◁
mutable afs_exit: fun } ▷ *Things to do on exit* ◁

[status] = **afs_server.afs_read_file** ~file : afs_file →

~off : int → ▷ *logical file offset in bytes* ◁
~size : int → ▷ *size in bytes* ◁
~buf : buffer

[status] = **afs_server.afs_modify_file** ~file : afs_file →
~off : int → ▷ *logical file offset in bytes* ◁
~size : int → ▷ *size in bytes* ◁
~buf : buffer

[status] = **afs_server.afs_modify_size** ~file : afs_file →
~newsize : int ▷ *size in bytes* ◁

[status] = **afs_server.afs_commit_file** ~file : afs_file →
~flag : int ▷ *commit flag* ◁

[status ∗
afs_file] = **afs_server.afs_read_inode** ~obj : int

[status] = **afs_server.afs_create_inode** ~file : afs_file →
~final : bool ▷ *file size final?* ◁

[status] = **afs_server.afs_delete_inode** ~file : afs_file

[status] = **afs_server.afs_modify_inode** ~file : afs_file

[afs_super ∗
status] = **afs_server.afs_read_super** ()

[status] = **afs_server.afs_sync** ()

[status ∗
string] = **afs_server.afs_stat** ~obj : int

[*destroy*: bool] = **afs_server.afs_age** ~obj : int

[unit] = **afs_server.afs_touch** ~file : afs_file

[status] = **afs_server.afs_exit** ()

Internal Server functions

The following functions are used by higher layers of the file server supplied by the user. The *acquire_file* function must be called to get a file structure associated with the file object number. After this function call, the file object is locked untill the *release_file* function is called. Additionaly, this function flushes caches and commit files depending on the commit flag. The *get_freeobjnum* function is used to get a free file object number.

[status ∗
afs_file] = **acquire_file** ~server : afs_file →
~obj : int

[status] = **release_file** ~server : afs_server →
~file : afs_file →
~flag : int

[*newobjnum*: int] = **get_freeobjnum** afs_super

Server request functions

These functions are used by higher levels to serve client requests. All sizes and offsets are in byte units.
Requests modifying files must distinguish this two cases:

1. The file state = *FF_unlocked* → Modification is uncritical. All functions returning capabilities return the old file capability.
2. The file state = *FF_locked* →
   I. Create a new file.
   II. Copy the original content and do the modifications in the newly created file. The new capability is returned.

afs_req_size
AFS server size request. Returns size of a file in byte units.

afs_req_create
AFS server create request. This function creates a new file. It expects a private field from an already existing file or the super capability. It returns the capability of the new created file object. The buffer content and the size is only initial if the commit flag is not set.

afs_req_read
AFS server read request. This functions reads *size* bytes starting at the file offset *off* into the buffer (always starting with position 0).

afs_req_modify
AFS server Modify request. This request modfies *size* bytes starting at file offset *off*. This request can be used to append new content to the file.

afs_req_insert
AFS server Insert request. This request insert *size* bytes at file offset *off* in the file.

afs_req_delete
AFS server Delete request. This request deletes *size* bytes starting at file offset *off* and decreases the file size.

afs_req_destroy
This request destroys a file object.

afs_req_stat
Get statistical informations for a file and the file system. Only a valid file object capability is accepted here.

afs_req_sync
Flush all caches. Either the super capability or a valid file object capability is accepted.

afs_req_touch
Set the live time of a file object to the maximal livetime number.

afs_req_age
Age the live time of all currently reachable files. Files with livetime = 0 are destroyed.

PROGRAMMING INTERFACE

[status ∗
size: int] = **afs_req_size** ~server : afs_server →
~priv : privat

[status ∗
newcap: capability] = **afs_req_create** ~server : afs_server →
~priv : privat →
~buf : buffer → ▷ *initial content* ◁
~size : int → ▷ *initial file size* ◁
~commit : int

[status ∗
numread: int] = **afs_req_read** ~server : afs_server →
~priv : privat →
~buf : buffer →
~off : int → ▷ *file offset <= file size* ◁
~size : int ▷ *requested size <= file size* ◁

[status ∗
newcap: capability] = **afs_req_modify** ~server : afs_server →
~priv : privat →
~buf : buffer →
~off : int → ▷ *file offset <= filesize* ◁
~size : int → ▷ *mod size* ◁
~commit : int

[status ∗
newcap: capability] = **afs_req_insert** ~server : afs_server →
~priv : privat →
~buf : buffer →
~off : int → ▷ *file offset <= filesize* ◁
~size : int → ▷ *insert size* ◁
~commit : int

[status ∗

newcap: capability] = **afs_req_delete** ~server : afs_server →
~priv : privat →
~off : int → ▷ *file offset <= filesize* ◁
~size : int → ▷ *delete size* ◁
~commit : int

[status] = **afs_req_destroy** ~server : afs_server →
~priv : privat

[status ∗
stat: string] = **afs_req_stat** ~server : afs_server →
~priv : privat

[status] = **afs_req_sync** ~server : afs_server →
~priv : privat

[status] = **afs_req_touch** ~server : afs_server →
~priv : privat

[status] = **afs_req_age** ~server : afs_server →
~priv : privat

Module Dependencies

- *Amoeba*
- *Bytebuf*
- *Cmdreg*
- *Stderr*
- *Stdcom*
- *Mutex*
- *Afs_common*

This module provides a fully implemented server loop. Usually, this server loop sufficient.
Before this function can be called, a server structure *afs_server* must be created. Additionaly, a sempahore *sema* must be initialized. The main server function will wait on each server thread. After the server thread exits, it increments this semaphore. The input and output buffer sizes are commonly choosen with the *afs_REQBUFSZ* value. The value *nthreads* tells this server threads how many server threads at all were spawned.

PROGRAMMING INTERFACE

[status] = **server_loop** ~server : afs_server →
~sema : semaphore →
~nthreads : int →
~inbuf_size : int →
~outbuf_size : int

Module: Afs_cache	*ML-Library: server.cma*

This module provides a generic fixed size file cache related to the Atomic File System specifications.

- The file cache consists of several fixed size buffers, commonly a multiple of the disk block size.
- Parts of a file (this can be the real file data or an inode block) or the whole file is cached.
- Objects are identified by their unique object number.
- It's assumed that files are stored continguesly on disk. That means: logical offset <=> disk address * block_size + offset.
- Offsets start with number 0. Physical addresses are in blocks.
- All buffers from a cache object are stored in a list sorted with increasing offset.

All the single buffers are allocated on startup with the cache_create function.
On a request, first the cache must be lookuped for an already cached file object. If there is no such object, a new cache entry is created. Therefore, the cache_read and cache_write functions must always provide the logical disk offset (when file offset = 0), and the current state of the file.
If a read request (obj,off,size) arrives, the desired fragment (off,size) is read into the cache, but the size is enlarged to the cache buffer size and the offset is adjusted modulo to the buffer offset (as long as the file end is not reached). The requested fragment is then copied into the user target buffer.
If the block file offset and size is already available in the cache, the desired data is only copied into the buffer. If there are already parts of the request cached, only the missing parts are read into the cache.
Basic File Parameters and units:

- File Sizes: Logical, in bytes
- Disk addresses: Physical, in blocks [super.afs_bock_size]
- File offsets: Logical, in bytes
- Free/Used Clusters: Physical (both addr and size!), in blocks
- A File always occupy full blocks

PROGRAMMING INTERFACE *Types and structure*

type afs_cache_mode = Cache_R ▷ *Write through the cache* ◁
| Cache_RW ▷ *Lazy read and write caching* ◁

type afs_cache_state = Cache_Empty ▷ *Empty cache buffer* ◁
| Cache_Sync ▷ *Cache buffer is synced with disk* ◁
| Cache_Modified ▷ *not synced with disk* ◁

type fsc_buf = { **mutable** fsb_index: int ; ▷ *the index of this buffer* ◁
mutable fsb_buf: buffer ; ▷ *the data buffer* ◁

```
                 mutable fsb_off: int ;                      ▷ logical file offset [bytes] ◁
                 mutable fsb_size: int ;                     ▷ the real amount of cached data [bytes] ◁
                 mutable fsb_state: afs_cache_state ;        ▷ state of the buffer ◁
                 mutable fsb_lock: Mutex.t }                 ▷ buffer lock ◁

type fsc_entry = { mutable fse_objnum: int ;                 ▷ Object number. Must be unique! ◁
                   mutable fse_disk_addr: int ;              ▷ physical object address [blocks] ◁
                   mutable fse_disk_size: int ;              ▷ physical object size [Bytes] ◁
                   mutable fse_cached: int list ;            ▷ list of all cached buffers [index] ◁
                   mutable fse_lastbuf: int ;                ▷ the last cached buffer [index] ◁
                   mutable fse_written: (int * int) list ;   ▷ list of already written bufs [LOFF,SI] ◁
                   mutable fse_state: afs_file_state ;       ▷ state of the fileobject ◁
                   mutable fse_live: int ;                   ▷ cache object live time ◁
                   mutable fse_lock: Mutex.t }               ▷ cache object lock ◁

type fsc_cache = { mutable fsc_size: int ;                              ▷ numbers of buffers in cache ◁
                   mutable fsc_block_size: int ;                        ▷ size of one buffer [bytes] ◁
                   mutable fsc_buf_size: int ;                          ▷ the buffer array ◁
                   mutable fsc_buffers: int list ;                      ▷ list of all free buffers ◁
                   mutable fsc_table: (int * fsc_entry) Hashtbl.t ;     ▷ obj buffer table ◁
                   mutable fsc_read: <fun> ;                            ▷ server suppiled disk read function ◁
                   mutable fsc_write: <fun> ;                           ▷ server supplied disk write function ◁
                   mutable fsc_synced: <fun> ;                          ▷ server supplied object-sync notify fun ◁
                   mutable fsc_lock: Mutex.t ;
                   mutable fsc_mode: afs_cache_mode ;                   ▷ the cache mode ◁
                   mutable fsc_stat: fsc_stat }                         ▷ cache statistics ◁
```

PROGRAMMING INTERFACE *Cache management*

```
[ status *
  fsc_cache ] = cache_create ~nbufs : int →             ▷ number of buffers ◁
                             ~blocksize : int →         ▷ blocksize of the filesystem [bytes] ◁
                             ~bufsize : int →           ▷ buffer size [blocks] ◁
                             ~read : <fun> →
                             ~write : <fun> →
                             ~sync : <fun> →
                             ~mode : afs_cache_mode     ▷ the cache mode ◁

[ unit ] = cache_compact ~cache : fsc_cache

[ found: bool *
  ce: fsc_entry ] = cache_lookup ~cache : fsc_cache →
                                 ~obj : int →               ▷ unique object number ◁
                                 ~addr : int →              ▷ disk address or other ◁
                                 ~size : int →              ▷ file size ◁
                                 ~state : afs_file_state    ▷ file state ◁

[ unit ] = cache_release ~cache : fsc_cache →
```

~fse : fse_entry

[status] = **cache_read** ~cache : fsc_cache → ▷ *cache* ◁
~fse : fsc_entry → ▷ *cache object* ◁
~buf : buffer → ▷ *data buffer to read in* ◁
~off : int → ▷ *logical file offset* ◁
~size : int ▷ *size of the desired fragment* ◁

[status] = **cache_write** ~cache : fsc_cache → ▷ *cache* ◁
~fse : fsc_entry → ▷ *cache object* ◁
~buf : buffer → ▷ *data buffer to write from* ◁
~off : int → ▷ *logical file offset* ◁
~size : int ▷ *size of the fragment* ◁

[status] = **cache_delete** ~cache : fsc_cache →
~fse : fsc_entry

[status] = **cache_commit** ~cache : fsc_cache →
~fse : fsc_entry

[status] = **cache_sync** ~cache : fsc_cache

[status] = **cache_age** ~cache : fsc_cache

[*stats*: string] = **cache_stat** ~cache : fsc_cache

This module contains generic types and structures used by the virtual disk server only. Each virtual disk is described by a disklabel structure, containing physical partition informations. Several physical partitions and disks can form one virtual disk.

Partition structure

p_firstblk
Gives the first physical block number of the partition.

p_numblks
Gives the number of physical blocks allocated to this partition.

p_piece
Each partition forms a part of exactly one virtual disk. p_piece specifies the number of the part that this partition is.

p_many
p_many specifies the total number of parts that make up the virtual disk.

p_cap
is the capability of the virtual disk to which this partition belongs.

p_sysid
system id string (aka. partition type name: 'Linux',...)

Disklabel structure

d_disktype
The type of this disk: Physical, Logical or Amoeba vdisk.

d_geom
Physical disk geometry informations of the disk to which this partition belongs.

d_ptab
Partition table array.

Virtual Disk structure

v_name
The name of the virtual disk: 'vdisk:01',...

v_cap
The capability of this virtual disk. The server port is the private (getport) one!

v_numblks
Virtual disk size: number of blocks of the virtual disk.

v_many
How many physical partitions form this virtual disk? Number of entries in the v_parts array.

v_parts
The physical partition array.

i_physpart
Partition struct of the physical partition on which the sub-partition is found.

i_part
The number of the partition (on the physical disk) from which this piece of virtual disk is taken.

i_firstblk
Number of the first block in this partition available for use by clients.

i_numblks
Number of blocks, starting from i_firstblk, available for use by clients.

Disk device structure

dev_host
The amoeba name of the host to which this device belongs to.

dev_path
The UNIX device path.

dev_blksize
Device block size in bytes.

dev_geom
Device geometry informations. Either derived from Amoeba disklabels or user/system suppield.

dev_read/dev_write
Device block read and write routines.

dev_labels
All the disklabels found on this device.

PROGRAMMING INTERFACE *Types and Structures*

val d_PHYS_SHIFT: int ▷ *Physical disk block size in log2 bits* ◁

val d_PHYSBLKSZ: int ▷ *Physical disk block size in bytes* ◁

val max_PARTNS_PDISK: int ▷ *maximum # of Amoeba partitions per physical disk* ◁

val disk_REQBUFSZ: int ▷ *RPC request/reply buffer size* ◁

```
type part_tab = { mutable p_firstblk: int32 ;
                  mutable p_numblks: int32 ;
                  mutable p_piece: int16 ;
                  mutable p_many: int16 ;
                  mutable p_cap: capability ;
                  mutable p_sysid: string }

type disktype =   Disk_physical of string
                | Disk_logical of string
                | Disk_amoeba of string
```

```
type disklabel = { mutable d_disktype: disktype ;
                   mutable d_geom: pdisk_geom ;
                   mutable d_ptab: part_tab array ;
                   mutable d_magic: int32 ;
                   mutable d_chksum : int32 }

type vpart = { mutable i_physpart: part_option ;    ▷ Virtual Disk Partition Table Structure ◁
               mutable i_part: int ;
               mutable i_firstblk: int ;
               mutable i_numblks: int }

type vdisk = { mutable v_name: string ;             ▷ Virtual Disk Structure ◁
               mutable v_cap: capability ;
               mutable v_numblks: int ;
               mutable v_many: int ;
               mutable v_parts: vpart array }

type dev_geom = { mutable dev_numcyl: int ;
                  mutable dev_numhead: int ;
                  mutable dev_numsect: int }

type disk_dev = { mutable dev_host: string ;
                  mutable dev_path: string ;
                  mutable dev_blksize: int ;
                  mutable dev_geom: dev_geom ;
                  mutable dev_read: <fun> ;
                  mutable dev_write: <fun> ;
                  mutable dev_labels: disklabel list }

val dev_read: first: int →
              size: int →
              buf: buffer →
              off: int →
              status

val dev_write: first: int →
               size: int →
               buf: buffer →
               off: int →
               status
```

This module implements the PC86 system dependent parts used by the virtual disk server.

This module contains the implementation of the virtual disk server.

vdisk_init
Initializes one disk:

1. Read physical (host) partition table.
2. Search for amoeba partitions, read the amoeba disklabels (subpartition table).
3. Create and initialize all virtual disk structures.

vdisk_table
Create one huge virtual disk table containing vdisks, pdisks and ldisks.

1. The real virtual amoeba disks (with highest vdisk#).
2. Physical disks (objnum = vdisk#+1...)
3. Logical disks.

The capabilities of the 2. & 3. disks must recomputed with respect to their object numbers!

vdisk_publish
Publish virtual disk capabilities (in directory dirname) somewhere in the DNS tree.
Note: the virtual disk structure (v_cap) holds the private (get) port!

vdisk_remove
Remove published vdisk caps. Should be done on server exit.

vdisk_rw
Disk read and write function. The oepration depends on the h_command field of the suppield header structure. Returns STD_OK if it could successfully read/write "vblksz" ∗ "num_vblks" bytes beginning at disk block "start" from/to the virtual disk specified by "priv" into/from "buf". Otherwise it returns an error status indicating the nature of the fault. The virtual disk capability is supplied in the private field of the header structure and is checked before the real operation.

vdisk_info
This routine returns in 'buf' the disk partition startblock and size for each physical disk partition comprising a virtual disk. The information was calculated at boot time and stored in network byte order at initialisation time. hdr.h_size is modified.

vdisk_size
Returns STD_CAPBAD if the capability was invalid or referred to a virtual disk with size <= 0. Otherwise it returns STD_OK and returns in "maxblocks" the maximum number of virtual blocks of size "2"l2vblksz" that fit on the virtual disk specified by "priv". Returns status and size.

vdisk_getgeom
Returns disk geometry informations. Because the geometry information in 386/AT bus machines is not on the disk but in eeprom the fdisk program cannot get at it. Therefore we provide this ugly hack to let it get it. hdr.h_size is modified.

vdisk_std_info
Returns standard string describing the disk server object. This includes the size of the disk in kilobytes. hdr.h_size is modified.

vdisk_std_restricts

This functions implements the STD_RESTRICT command. There is only one rights bit anyway so it isn't too exciting. The new private part is stored in hdr.h_priv.

PROGRAMMING INTERFACE

[status] = **vdisk_init** ~dev : disk_dev →
~disks : disk

[vdisk option array] = **vdisk_table** disk

[status] = **vdisk_publish** ~disk_table : vdisk option array →
~dirname : string

[status] = **vdisk_remove** ~disk_table : vdisk option array →
~dirname : string

[status] = **vdisk_rw** ~hdr : header →
~vblksz : int →
~num_vblks : int →
~buf : buffer →
~vdisks : vdisk option array

[status] = **vdisk_info** ~hdr : header →
~buf : buffer →
~vdisks : vdisk option array

[status *
size: int] = **vdisk_size** ~hdr : header →
~vblksz : int →
~vdisks : vdisk option array

[status] = **vdisk_getgeom** ~hdr : header →
~buf : buffer →
~vdisks : vdisk option array

[status] = **vdisk_std_info** ~hdr : header →
~buf : buffer →
~vdisks : vdisk option array

[status] = **vdisk_std_restrict** ~hdr : header →
~rights : int →
~vdisks : vdisk option array

Module dependencies

- ➤ Amoeba
- ➤ Buffer
- ➤ Disk_common

This module implements the rpc server loop of the virtual disk server. Several threads can be started with this function.

disk_table
The virtual disk table.

sema
Semaphore used to synchronosize the master server. After this service thread exited, the semaphore is incremented.

nthreads
Total number of service threads.

inbuf_size
Request buffer size. Usually equal to disk_REQBUFSZ.

outbuf_size
Reply buffer size. Usually equal to disk_REQBUFSZ.

vblksize
Blocksize of the virtual disk server (not necessary the physical block size!).

PROGRAMMING INTERFACE

[unit] = **server_loop** ~disk_table : vdisk option array →
~sema : semaphore →
~nthreads : int →
~inbuf_size : int →
~outbuf_size : int →
~vblksize : int

EXAMPLE *UNIX Virtual Disk Server*

```
open Amoeba
open Stderr
open Unix
open Disk_common
open Disk_server
open Disk_server_rpc
open Bytebuf
open Printf
open Machtype
```

```
open Buffer
open Sema
open Thread

let path = "/dev/da0"
let nl = print_newline

let init () =
   let blksize = d_PHYSBLKSZ in
   let disk_fd = Unix.openfile path [O_RDWR] 0 in
   let hostname = Unix.gethostname () in

   (*
   ** Reads # size of blocks from disk starting at off block.
   *)
   let read ~first ~size ~buf ~off =
   try
   begin
      print_string (sprintf "read:  first=%d size=%d boff=%d"
                     first size off); nl ();

      let off' = first*blksize in
      let size' = size*blksize in
      let n = lseek disk_fd off' SEEK_SET in
      if n <> off' then
         raise (Error std_IOERR);
      let n = readb disk_fd buf off size' in
      if n <> size' then
         raise (Error std_IOERR);
      std_OK
   end
      with | Error err -> err
         | _ -> std_IOERR
      in

   (*
   ** Write # size of blocks to disk starting at off block.
   *)
   let write ~first ~size ~buf ~off =
   try
   begin
      print_string (sprintf "write:  first=%d size=%d boff=%d"
                     first size off); nl ();

      let off' = first*blksize in
      let size' = size*blksize in
      let n = lseek disk_fd off' SEEK_SET in
      if n <> off' then
         raise (Error std_IOERR);

      let n = writeb disk_fd buf off size' in
      if n <> size' then
         raise (Error std_IOERR);
```

```
        std_OK
    end
        with | Error err → err
            | _ → std_IOERR
        in


    let dev = {
            dev_host = "/hosts/"^hostname;
            dev_path = path;
            dev_blksize = blksize;
            dev_read = read;
            dev_write = write;
            dev_labels = [];
            dev_geom = {dev_numcyl = 0;
                        dev_numhead = 0;
                        dev_numsect = 0};
        } in
    let disks = {
            pdisks = [];
            vdisks = [];
            ldisks = [] } in

    let stat = vdisk_init ~dev:dev ~disks:disks in
    printf "disk_init:  %s" (err_why stat);
    nl ();
    dev,disk_fd,disks


let _ =
    let dev,disk_fd,disks = init () in
    let vt = vdisk_table disks in

    (*
    let str = info_vdisks dev disks in
    print_string str ;
    *)

    let str = print_vdisk_table vt in
    print_string str;

    let stat = vdisk_publish vt "/hosts/develop" in
    printf "vdisk_publish:  %s\n" (err_why stat);

    (*
    ** Start service threads
    *)
    let sem = sema_create 0 in
    let nthr = 4 in

    printf "Starting service threads..."; nl();


    for i = 1 to nthr
```

```
do
   ignore (thread_create (fun () → server_loop vt
                                                sem
                                                nthr
                                                disk_REQBUFSZ
                                                disk_REQBUFSZ
                                                d_PHYSBLKSZ) ());

done;
for i = 1 to nthr
do
   sema_down sem;
done;
Unix.close disk_fd;
print_string "bye.."; nl ();
```

The DNS server : Directory and Name Service.
Common values and structures - both for servers and clients:

Requests and Rights

PROGRAMMING INTERFACE

val dns_CREATE: command

val dns_DISCARD: command

val dns_LIST: command

val dns_APPEND: command

val dns_CHMOD: command

val dns_DELETE: command

val dns_LOOKUP: command

val dns_SETLOOKUP: command

val dns_INSTALL: command

val dns_REPLACE: command

val dns_GETMASKS: command

val dns_GETSEQNR: command

val dns_RGT_DEL: int

val dns_RGT_MOD: int

dns_CREATE
Create a directory. The request must supply the names of the columns. By counting this name list, the server know the number of coulumns!

dns_DISCARD
Destroy a directory. Only allowed if the directory is empty.
Required rights: *dns_RGT_DEL*

dns_LIST
List a directory. The request returns a flattened representation of the number of columns, the number of rows, the names of the columns, the names of the rows and the rights

masks. Perhaps not all rows can delivered within one request. Following requests are needed in this case.

dns_APPEND
Append a row to an already existing directory. The row name and the new capability must be delivered with this requets.
Required rights: *dns_RGT_MOD*

dns_CHMOD
Change the rights masks of a directory row.
Required rights: *dns_RGT_MOD*

dns_DELETE
Delete a row of a directory.
Required rights: *dns_RGT_DEL*

dns_LOOKUP
Traverse a path as far as possible, and return the resulting capability set and the unresolved part of the path.

dns_SETLOOKUP
Lookup rownames in a set of directories.

dns_INSTALL
Update a set of directory entries. All entries have to be at this directory server or it won't work. Specified are capability sets for directories (simple names, no pathnames). Moreover, an old capability ca be specified which has to be in the current capability set for the update succeeded.
Required rights: *dns_RGT_MOD*

dns_REPLACE
Replace a capability set. The name and the new capability set is specified.
Required rights: *dns_RGT_MOD*

dns_GETMASKS
Return the rights masks in a row.

dns_GETSEQNR
Return the sequence number (request counter) of the directory.

dns_GETDEFAULTFS
Get the default file server capability, if any.

Functions

The *path_normalize* function evaluates any "." or ".." components in the path name, remove multiple '/' and evaluates relative paths. Returns the normalized path.

PROGRAMMING INTERFACE

[string] = **path_normalize** ~path : string

Module dependencies

- *Amoeba*
- *Cmdreg*

This is the DNS client module. It provides functions to lookup and modify existing directories, delete or extract rows (directory entries).

PROGRAMMING INTERFACE

[*rpc_status*: status ∗
cs: capset] = **lookup** ~root : capset →
~path : string

dns_LOOKUP

This request is used to get a capability set for a path relative to a root directory. A root directory can be any directory in the directory tree. The functions loops over the path components step by step and resolve the path.
In each run, the next path component is the new parent directory for the next lookup (to another DNS server).

Module dependencies

- *Amoeba*
- *Bytebuf*
- *Capset*
- *Cmdreg*
- *Stderr*
- *Dns_common*

This module provides the server side implementation of the DNS Directory and Name service. This module provides structures and basic functions to build a DNS server.

Basic structures

fs_server
The file server structure. DNS server objects (aka. directories) are saved as AFS files. Therefore at least one file server must be known. In one copy mode, only files are read and written from this server. In two copy mode, the directories are duplicated onto two file servers. If one file servers craches, the other can be still used.

dns_row
One row of a directory (= directory entry):

dns_dir
One DNS table entry (= directory). All rows are stored in a double linked list.

dns_super
The main DNS server structure: the all known super structure with fundamental informations about the DNS.

dns_server
The server structure. The *dns_read_dir* and *dns_write_dir* functions are used to read and write single directories. The *dns_create_dir* and *dns_delete_dir* functions are used to add and delete directories. The *dns_read_super* function is used to read the super structure of the file system. And finally, the *dns_sync* function is used to flush all caches of the server, if any. The implementation of these functions must be provided by higher levels outside of this module.

PROGRAMMING INTERFACE

```
type dns_mode =   Dnsmode_ONECOPY      ▷ Only one file server is used ◁
                | Dnsmode_TWOCOPY      ▷ Duplicated file server mode ◁

type fs_state =   FS_down        ▷ File server is down ◁
                | FS_up          ▷ File server is up ◁
                | FS_unknown     ▷ Don't know ◁

type fs_server = { mutable fs_cap: capability array ;
                   mutable fs_state: fs_state array ;
                   mutable fs_default: int ;           ▷ The default file server ◁
                   mutable dns_mode: dns_mode }         ▷ DNS server mode ◁

type dns_row = { mutable dr_name: string ;                 ▷ The row name ◁
                 mutable dr_time: int ;                    ▷ Time stamp ◁
                 mutable dr_columns: rights_bits array ;   ▷ The rights mask ◁
                 mutable dr_capset: capset }               ▷ Row capability set of the object ◁

type dns_dir_state =   DD_invalid      ▷ Not used ◁
```

```
                      | DD_unlocked     ▷ New: Can be still modified ◁
                      | DD_modified     ▷ Previously locked; now modified ◁
                      | DD_locked       ▷ Written to disk: valid ◁

type dns_dir = { mutable dd_lock: Mutex.t ;                        ▷ Directory lock ◁
                 mutable dd_objnum: int ;                          ▷ The directory index number ◁
                 mutable dd_ncols: int ;                           ▷ Number of columns ◁
                 mutable dd_nrows: int ;                           ▷ Number of rows in this directory ◁
                 mutable dd_colnames: string array ;               ▷ The columns names ◁
                 mutable dd_random: port ;                         ▷ Random check number ◁
                 mutable dd_rows: (dns_row dblist) option ;        ▷ The rows ◁
                 mutable dd_state: dns_dir_state ;                 ▷ Status of the directory ◁
                 mutable dd_time: int ;                            ▷ Time stamp ◁
                 mutable dd_live: int }                            ▷ Live tim of object ◁

type dns_super = { mutable dns_lock: Mutex.t ;
                   mutable dns_name: string ;                                 ▷ The server name/label ◁
                   mutable dns_ndirs: int ;                                   ▷ Number of total table entries ◁
                   mutable dns_nused: int ;                                   ▷ Number of used table entries ◁
                   mutable dns_freeobjnums: int list ;                        ▷ Free slots list ◁
                   mutable dns_nextfree: int ;                                ▷ Next free slot ◁
                   mutable dns_getport: port ;                                ▷ Private server port ◁
                   mutable dns_putport: port ;                                ▷ Public server port ◁
                   mutable dns_checkfield: port ;                             ▷ Private checkfield ◁
                   mutable dns_ncols: int ;                                   ▷ Number of columns ◁
                   mutable dns_colnames: string array ;                       ▷ Column names ◁
                   mutable dns_generic_col_mask: rights_bits array ;          ▷ Column mask ◁
                   mutable dns_fs_server: fs_server ;                         ▷ File server used ◁
                   mutable dns_block_size: int }                              ▷ Blocksize in bytes ◁

type dns_server = { mutable dns_super: dns_super option ;         ▷ The super structure ◁
                    mutable dns_read_dir: fun ;
                    mutable dns_modify_dir: fun ;
                    mutable dns_create_dir: fun ;
                    mutable dns_delete_dir: fun ;
                    mutable dns_read_super: fun ;
                    mutable dns_sync: fun ;
                    mutable dns_stat: fun ;
                    mutable dns_touch: fun ;
                    mutable dns_age: fun ;
                    mutable dns_exit: fun }

[ dns_dir *
  status     ] = dns_server.dns_read_dir ~obj : int

[ status ] = dns_server.dns_modify_dir ~dir : dns_dir

[ status ] = dns_server.dns_create_dir ~dir : dns_dir

[ status ] = dns_server.dns_delete_dir ~dir : dns_dir

[ dns_super *
  status       ] = dns_server.dns_read_super ()
```

[status] = **dns_server.dns_sync** ()

[status *
string] = **dns_stat** ()

[status] = **dns_server.dns_touch** ~dir : dns_dir

[*used*: bool *
time: int] = **dns_server.dns_age** ~obj : int

[status] = **dns_server.dns_exit** ()

Values

The *dns_col_bits* array specifies the rights bits for each columns. Indead, only the first *dns_MAXCOLUMNS* array elements are used. The *dns_MAXLIVE* value specifies the maximal live time of DNS objects (touch/age mechanism).

PROGRAMMING INTERFACE

val dns_MAXLIVE: int

val dns_col_bits: rights_bits array

Internal functions

The *acquire_dir* function must be called before a client request can be handled. It returns the directory structure and locks the directory. After the rqeuest is finished, the *release_dir* must be called to unlock the directory and to perfom pending write operations if any. The *get_dir* function returns the private field of a directory capability set.

PROGRAMMING INTERFACE

[*dir*: dns_dir *
err: status] = **acquire_dir** ~server : dns_server →
~priv : privat →
~req : rights_bits

[unit] = **release_dir** ~server : dns_server →
~dir : dns_dir

[*err*: status ∗
priv: privat] = **get_dir** ~server : dns_server →
~dir_cs : capset

Directory table management

dns_search_row
This functions searches a directory for the row given row name. It returns the row on success, else *None* if the directory doesen't have such a row name.

dns_create_dir
This function creates a new directory. the function returns the new directory structure and teh status returned by the servers *dns_dir_create* function.

dns_delete_dir
Delete an empty directory. The servers *dns_delete_dir* function is called.

dns_destroy_dir
Delete a directory, no matter if empty or not. The servers *dns_delete_dir* function is called.

dns_create_row
This function creates a new row.

dns_append_row
This function appends a newly created row to an existing directory.

dns_delete_row
Delete a row of an existing directory.

capset_of_dir
This function returns a capability set with one capability derived from the directory with a private field encoded with the *rights* mask.

dns_restrict
This function creates a restricted version of either a directory capability or of the object from other server. The restricted capability is created with the new rights from the mask value. For foreign objects, the *std_restrict* function is used to fullfill the restriction.

PROGRAMMING INTERFACE

[*row*: dns_row option] = **dns_search_row** ~dir : dns_dir →
~name : string

[*dir*: dns_dir ∗
err: status] = **dns_create_dir** ~server : dns_server

[*err*: status] = **dns_delete_dir** ~server : dns_server →
~dir : dns_dir

[*err*: status] = **dns_destroy_dir** ~server : dns_server →

~dir : dns_dir

[*err*: status] = **dns_create_row** ~name : string →
~cols : rights_bits_array →
~cs : capset

[*err*: status] = **dns_append_row** ~server : dns_server →
~dir : dns_dir →
~row : dns_row

[*err*: status] = **dns_delete_row** ~server : dns_server →
~dir : dns_dir →
~row : dns_row

[*cs*: capset] = **capset_of_dir** ~super : dns_super →
~dir : dns_dir →
~rights : rights_bits

[*err*: status ∗
cs_restr: capset] = **dns_restrict** ~server : dns_server →
~cs_orig : capset →
~mask : rights_bits

Client request handlers

dns_req_lookup
Traverse a path as far as possible, and return the resulting capability set and the rest of the unresolved path fragment.

dns_req_list
List the content of a directory. returns a flattend representation of the number of columns, the number of rows, the names of the columns, the names of the rows and the right masks. The rows are returned in a *(dr_name,dr_columns)* tuple list.

dns_req_append
Append a row to an already existing directory. The name, right masks (cols), and the initial capability must be specified.

dns_req_create
Create a new directory table entry.

dns_req_discard
Remove a directory from the table. Simple. Only allowed if the directory is empty. The *dns_DELRGT* rights are required to perform this operation.

dns_req_destroy
Destroy a directory from the table. Simple. Allowed for empty and not empty directories. The *dns_DELRGT* rights are required to perform this operation.

dns_req_chmod
Change the rights masks in a row. The *dns_MODRGT* rights are needed.

dns_req_delete
Delete a row within a directory. The *dns_MODRGT* rights are required to perform this operation.

dns_req_replace

Replace a capability set. The row name and new capability set must be specified. The *dns_MODRGT* rights are required to perform this operation.

dns_req_touch
Set the live time of the specified directory to the maximal value.

dns_req_age
Age all DNS objects. All valid object with live time zero will be destroyed. Only allowed with the unrestricted super capability.

dns_req_setlookup
Lookup rownames in a set of directories. The *dirs* argument is a list of *(dir_cs,rowname)* tuples. The function returns the resolved rows list with *(status,time,capset)* tuples.

PROGRAMMING INTERFACE

```
[ err: status *
  cs: capset *
  path_rest: string ] = dns_req_lookup ~server : dns_server →
                                       ~priv : privat →
                                       ~path : string

[ err: status *
  nrows: int *
  ncols: int *
  colnames: string array *
  (string * rights_bits array) list ] = dns_req_list ~server : dns_server →
                                                     ~priv : privat →
                                                     ~firstrow : int

[ err: status ] = dns_req_append ~server : dns_server →
                                 ~priv : privat →
                                 ~name : string →
                                 ~cols : rights_bits_array →
                                 ~capset : capset

[ err: status *
  newcs: capset ] = dns_req_create ~server : dns_server →
                                   ~priv : privat →
                                   ~colnames : string array

[ err: status ] = dns_req_discard ~server : dns_server →
                                  ~priv : privat

[ err: status ] = dns_req_destroy ~server : dns_server →
                                  ~priv : privat

[ err: status ] = dns_req_chmod ~server : dns_server →
                                ~priv : privat →
                                ~cols : rights_bits array →
                                ~name : string
```

[*err*: status] = **dns_req_delete** ~server : dns_server →
~priv : privat →
~name : string

[*err*: status] = **dns_req_replace** ~server : dns_server →
~priv : privat →
~name : string →
~newcs : capset

[*err*: status] = **dns_req_age** ~server : dns_server →
~priv : privat

[*err*: status] = **dns_req_touch** ~server : dns_server →
~priv : privat

[*err*: status ∗
cols: rights_bits array] = **dns_req_getmasks** ~server : dns_server →
~priv : privat →
~name : string

[(status∗int∗capset) list] = **dns_req_setlookup** ~server : dns_server →
~dirs : (capset ∗ string) list

Module dependencies

- *Amoeba*
- *Bytebuf*
- *Capset*
- *Cmdreg*
- *Dblist*
- *Mutex*
- *Stderr*
- *Dns_common*

This module provides a generic server loop for the DNS server. The *inbuf_size* and *outbuf_size* arguments specify the size of the request and reply transaction buffers. Commonly set to *dns_BUFSIZE*.

PROGRAMMING INTERFACE

```
[ unit ] = server_loop ~server : dns_server →
                       ~sema : semaphore →        ▷ server wait sema ◁
                       ~nthreads : int →          ▷ number of server threads ◁
                       ~inbuf_size : int →        ▷ request buffer size ◁
                       ~outbuf_size               ▷ reply buffer size ◁
```

Module dependencies

- *Amoeba*
- *Bytebuf*
- *Capset*
- *Cmdreg*
- *Dblist*
- *Mutex*
- *Rcp*
- *Stdcom*
- *Stdcom2*
- *Stderr*
- *Thread*
- *Dns_common*
- *Dns_server*

Example

```
(*
** A simple DNS server example.  Directories are not saved to disk.
*)

open Amoeba
open Stdcom
open Stderr
open Thread
open Rpc
open Capset
```

```
open Dns_common
open Dns_server
open Dns_server_rpc

let ndirs = 1000
let ncols = 3
let colnames = [|"owner";"group";"world"|]
let stdcolmask = [|Rights_bits 0xff;
             Rights_bits 0x2;
             Rights_bits 0x4 |]


let init ()    =
   let refnildnsdir = ref nildnsdir in

   let super = dns_create_super ~name:"example"
                        ~ndirs:ndirs
                        ~ncols:ncols
                        ~colnames:colnames
                        ~colmask:stdcolmask
   in

   let ic = open_in "/home/sbosse/.dns_super" in
   let (super':dns_super) = input_value ic in
   close_in ic;

   super.dns_getport <- super'.dns_getport;
   super.dns_putport <- super'.dns_putport;
   super.dns_checkfield <- super'.dns_checkfield;

   let dir_table = Array.create ndirs refnildnsdir in

   (*
   ** Server functions.  Mostly dummies because the server don't save
   ** directories to disk!
   *)

   let read_dir ~objnum =
      !(dir_table.(objnum)),std_OK
      in

   let write_dir ~dir =
      dir.dd_state <- DD_cached;
      std_OK
      in

   let create_dir ~dir =
      let rdir = ref dir in
      dir_table.(dir.dd_objnum) <- rdir;
      std_OK
      in

   let delete_dir ~dir =
      dir_table.(dir.dd_objnum) <- refnildnsdir;
```

```
        std_OK
        in

    let read_super () =
        super,std_OK
        in

    let write_super ~super     =
        std_OK
        in

    let sync () =
        std_OK
        in

    let server = {
        dns_lock = mu_create ();
        dns_super = super;
        dns_read_dir = read_dir;
        dns_write_dir = write_dir;
        dns_create_dir = create_dir;
        dns_delete_dir = delete_dir;
        dns_read_super = read_super;
        dns_write_super = write_super;
        dns_sync = sync;
        }
        in
    let stat,root = dns_create_root ~server:server in

    server,dir_table,root


(*
** The server itself.
*)

let server () =
    let server,table,root = init () in

    (*
    ** Create some example directories and entries!
    *)

    let s1,d1 = dns_create_dir ~server:server ~ncols:ncols
                        ~colnames:colnames
    in
    let s2,d2 = dns_create_dir ~server:server ~ncols:ncols
                        ~colnames:colnames
    in
    let s3,d3 = dns_create_dir ~server:server ~ncols:ncols
                        ~colnames:colnames
    in
    let cs1 = capset_of_dir ~super:server.dns_super
                        ~dir:d1
```

```
                    ~rights:prv_all_rights

in

let r1 = dns_create_row ~name:"testdir1"
                    ~cols:[|Rights_bits 0xff;
                           Rights_bits 0x2;
                           Rights_bits 0x4|]
                    ~cs:cs1
in
let cs2 = capset_of_dir ~super:server.dns_super
                    ~dir:d2
                    ~rights:prv_all_rights

in

let r2 = dns_create_row ~name:"up"
                    ~cols:[|Rights_bits 0xff;
                           Rights_bits 0xf6;
                           Rights_bits 0x4|]
                    ~cs:cs2
in
let cs3 = capset_of_dir ~super:server.dns_super
                    ~dir:d3
                    ~rights:prv_all_rights

in

let r3 = dns_create_row ~name:"down"
                    ~cols:[|Rights_bits 0xff;
                           Rights_bits 0x2;
                           Rights_bits 0x4|]
                    ~cs:cs3
in
ignore( dns_append_row ~server:server
                    ~dir:root
                    ~row:r1
);
ignore( dns_append_row ~server:server
                    ~dir:d1
                    ~row:r2
);
ignore( dns_append_row ~server:server
                    ~dir:d1
                    ~row:r3
);

(*
** The root directory.
*)
let super = server.dns_super in
let cs = capset_of_dir ~super:super
                ~dir:root
                ~rights:prv_all_rights
```

```
                 in
    let putcap = cs.cs_suite.(0).s_object in


    Printf.printf "%s" (Ar.ar_cap putcap);
    print_newline ();

    server_loop ~server:server
                ~inbuf_size:30000
                ~outbuf_size:30000


let _ =
    server ()
```

Object manager server → Garbage collection. This module implemts basic concepts for recursive touchung of reachable objects in an Amoeba directory and filesystem. Also, it initaites aging of unreachable/unused objects.

Setup configuration, initialization and looping

First a defintion list with entries of type *om_config* must be build and initialized with the function *om_init*. After, the real work is done in *om_loop*.

PROGRAMMING INTERFACE

type om_config = Om_root of string ▷ *root directory path>* ◁
| Om_root_cap of string ▷ *same, but cap. specified* ◁
| Om_ignore of string ▷ *don't itereate this path* ◁
| Om_ignore_cap of string ▷ *same, but cap. in Ar format* ◁
| Om_age of string ▷ *age this server spec. with path* ◁
| Om_age_cap of string ▷ *same, but cap. in Ar format* ◁
| Om_mintouch of int ▷ *min. successfull touched objs. before aging* ◁

type om = { **mutable** om_root: capability ; ▷ *start root directory* ◁
mutable om_ingore: capability list ; ▷ *dirs. to ignore* ◁
mutable om_age: capability list ; ▷ *age all objects from spec. servers* ◁
mutable om_mintouch: int ; ▷ *min. touched objs. before aging is started* ◁
mutable om_touched: int ; ▷ *touched objects* ◁
mutable om_failed: int ; ▷ *not reachable objects* ◁
mutable om_report: string ; ▷ *report string* ◁
mutable om_passnow: int ; ▷ *current pass number* ◁
mutable om_passnum: int } ▷ *maximal number of passes* ◁

[om] = **om_init** om_config list

[status] = **om_loop** om

Example

```
open Name ;;
open Om ;;

let conf = [
   Om_root "/";
   Om_ignore "/hosts";
   Om_passnum 10;
   Om_age "/server/afs";
   Om_mintouch 10;
```

```
]
let om = Om.om_init conf ;;
Om.om_loop om ;;
```

A simple boot service implementation. This module provides basic functions and types to build up boot services.
Different kinds of boot objects are supported, with several possible object source locations and specifiers. Also, the destination for a boot object can be specified.
With each object, one or several boot operations can be specified. Each boot object can hold his own execution environment.

Boot loaction

PROGRAMMING INTERFACE

```
type boot_loc =   Obj_file_path of Cap_env.path_arg    ▷ Either UNIX or Amoeba file ◁
                | Obj_cap of string                    ▷ Amoeba capability in Ar format ◁
                | Obj_cap_path of Cap_env.path_arg     ▷ Either UNIX or AMoeba cap. ◁
```

Source file location

Where to get the binary or script file content.

PROGRAMMING INTERFACE

```
type boot_src =   Binary_src of boot_loc    ▷ Unix or Amoeba file system ◁
                | Buf_src of string         ▷ The source is a buffer content ◁
                | Nil_src
```

Boot object destination system

This type specifies the execution host system: native Amoeba, local UNIX system, machine independent bytecode system.

PROGRAMMING INTERFACE

```
type boot_dst =   Unix_dst                  ▷ executes on Unix ◁
                | Amoeba_dst of boot_loc    ▷ path for Amoeba process server ◁
```

```
                | ML_dst of boot_loc        ▷ ML interpreter program path ◁
                | VM_dst of boot_loc        ▷ Forth script interpreter ◁
                | Nil_dst
```

Boot object status and operations - the environmnt

At least one operation must be done for a boot object, for example one time execution of the binary program on the given host environment. The started boot object can be polled (still alive?), and in the case the *Boot_restart* operation is specified, the service is restarted on crash.

PROGRAMMING INTERFACE

```
type boot_op =    Boot_poll of (boot_loc * int)    ▷ poll path/cap and time interval ◁
                | Boot_start                       ▷ start the service ◁
                | Boot_stop                        ▷ stop the service ◁
                | Boot_restart                     ▷ restart if polled and crashed ◁
                | Boot_fun of (unit → unit)        ▷ execute an internal fun. ◁

type boot_stat =    Boot_starting
                  | Boot_killing
                  | Boot_up
                  | Boot_down
                  | Boot_restarting
                  | Boot_unknown

type boot_env =    Env_cap of (string * capability)    ▷ Amoeba environment ◁
                 | Env_arg of (string * string)        ▷ UNIX environment ◁
                 | Env_self of string                  ▷ Handled by the boot server ◁
```

Boot object descriptor

One boot object is handled with the following descriptor structure with all necessary information about the service.

PROGRAMMING INTERFACE

```
type boot_def = {  boot_name:string ;        ▷ name of the boot object ◁
                   boot_mode:boot_exe ;      ▷ the boot object type ◁
                   boot_src:boot_src ;       ▷ the boot object source ◁
                   boot_dst:boot_dst ;       ▷ the boot execution eivironment ◁
```

```
boot_args:string list ;         ▷ program arguments ◁
boot_env:boot_env list ;        ▷ program cap. env. ◁
boot_ops:boot_op list ;         ▷ boot obj. operations ◁
boot_deps:string list }         ▷ boot obj. name dependencies ◁
```

Boot object public interface

To build up a boot server, you only need to build the boot definition structure array and call the *boot_init* function. This funtions returns the internal bootserver structure. Just call *boot_start* function to initialize and coldstart (maybe with previous poll check) all specified boot objects. After this, a dedicated service loop can be started with the *boot_loop* function (starts a new thread!). To wait for the shutdown of the boot server, just call the *boot_wait* function.

PROGRAMMING INTERFACE

[*bs*: boot_server ∗
stat: status] = **boot_init** *defs*: boot_def array

[status] = **boot_start** *bs*: boot_server

[string ∗
status] = **boot_status** *bs*: boot_server

[status] = **boot_stop** *bs*: boot_server

[status] = **boot_loop** *bs*: boot_server

[status] = **boot_loop** *bs*: boot_server

[status] = **boot_wait** *bs*: boot_server

Example

```
(*
**    Default VAM boot script.
*)


open Amoeba ;;
open Vamboot ;;
open Unix ;;
open Cap_env ;;
open Sema ;;
open Stderr ;;
```

```
open Stdcom ;;

let supercap = ref nilcap ;;

let defs = [|
   { boot_name = "flipd";
   boot_mode = Unix_exe;
   boot_src = Binary_src (Obj_file_path (
               Unix_path "/amoeba/Amunix/bin/flipd"));
   boot_dst = Unix_dst;
   boot_args = [];
   boot_env = [];
   boot_ops = [Boot_start;Boot_stop];
   boot_deps = [];
   };
   { boot_name = "afs_unix";
   boot_mode = Unix_exe;
   boot_src = Binary_src (Obj_file_path (
               Unix_path "/amoeba/Vam-1.7/bin/afs_unix"));
   boot_dst = Unix_dst;
   boot_args = ["-s"];
   boot_env = [];
   boot_ops = [Boot_start;
            Boot_stop;
            Boot_poll (Obj_cap_path (
               Unix_path "/amoeba/afs/.servercap"),5);
            ];
   boot_deps = ["flipd"];
   };
   { boot_name = "dns_unix";
   boot_mode = Unix_exe;
   boot_src = Binary_src (Obj_file_path (
               Unix_path "/amoeba/Vam-1.7/bin/dns_unix"));
   boot_dst = Unix_dst;
   boot_args = ["-s"];
   boot_env = [];
   boot_ops = [Boot_start;
            Boot_stop;
            Boot_poll (Obj_cap_path (
               Unix_path "/amoeba/dns/.servercap"),5);
            ];
   boot_deps = ["flipd";"afs_unix"];
   };
   { boot_name = "xafs";
   boot_mode = Unix_exe;
   boot_src = Binary_src (Obj_file_path (
               Unix_path "/amoeba/Vam-1.7/bin/xafs"));
   boot_dst = Unix_dst;
   boot_args = [""];
   boot_env = [];
   boot_ops = [Boot_start;
            Boot_stop;
            ];
   boot_deps = ["flipd";"afs_unix";"dns_unix"];
```

```
    };
    { boot_name = "boot_pub";
    boot_mode = Fun_exe;
    boot_src = Nil_src;
    boot_dst = Nil_dst;
    boot_args = [];
    boot_env = [];
    boot_ops = [
                 (Boot_fun (fun () →
                    ignore (Name.name_delete "/server/boot");
                    ignore (Name.name_append "/server/boot"
                                             !supercap);
                    boot_info ("supercap= "^(Ar.ar_cap !supercap));
                    ));
              ];
    boot_deps = ["flipd";"afs_unix";"dns_unix"];
    };
|];;

let bs,stat = boot_init defs ;;
supercap := bs.boot_supercap ;;

boot_start bs ;;
boot_loop bs ;;
boot_wait bs ;;
```

Portable thread module with mutex and semaphore support and thread synchronisation.

thread_create
Creates a new thread and returns the new thread id. Expects a user supplied function.

thread_exit
A thread terminates. If the user supplied function returns, this function is called automatically.

thread_id
Returns the thread id of the current ruinning thread.

thread_switch
Release control to the scheduler which can switch to another ready to run thread. If there are no other ready threads, the function just returns.

thread_sdelay
Delay the current thread for ## seconds.

thread_mdelay
Delay the current thread for ## milli seconds.

thread_udelay
Delay the current thread for ## micro seconds.

thread_delay
Delay teh current thread for at least ## UNIT seconds. The UNIT can be micro-, milliseconds or seconds. If this call is interrupted, the function returns false.

thread_await
A thread wants to wait for an event. The maximal await timeout interval in milli seconds is specified. If a timeout or an interrupt occurs, this function returns false.

thread_wakeup
Raise event ev and wakeup the next thread waiting for this event. If no thread is already waiting for this event, increment pending variable, so this wakeup call is not lost.

mu_create
Create a new locking mutual exclusion.

mu_lock
Lock a mutex. If the mutex is already locked, suspend current thread untill mutex is released by the owner.

mu_trylock
Try to lock a mutex. Returns false if the mutex is already locked.

mu_unlock
Unlock a mutex.

PROGRAMMING INTERFACE

[*tid*: int] = **thread_create** ~func : ('a→'b) →
~arg : 'a

[unit] = **thread_exit** ()

[int] = **thread_id** ()

[unit] = **thread_switch** ()

[thread_event] = **thread_create_event** ()

[bool] = **thread_await** thread_event →
interval: int

[unit] = **thread_wakeup** thread_event

type time_unit = SEC
| MILLISEC
| MICROSEC

[bool] = **thread_delay** int →
time_unit

[unit] = **thread_sdelay** *secs*: int

[unit] = **thread_mdelay** *msecs*: int

[unit] = **thread_udelay** *usecs*: int

[Mutex.t] = **mu_create** ()

[unit] = **mu_lock** Mutex.t

[bool] = **mu_trylock** Mutex.t

[unit] = **mu_unlock** Mutex.t

A thread may only lock one time a mutex, else an exception is raised because this is really a programmign error! Furthermore, only the owner therda may unlock a mutex. If another try to unlock a not owned lock, it will raise an exception, too.

PROGRAMMING INTERFACE

[Mutex.t] = **create** ()

[unit] = **lock** Mutex.t

[bool] = **try_lock** Mutex.t

[unit] = **unlock** Mutex.t

Semaphore Module: Thread synchronization using counting semaphores.
These operations implement counting semaphores. What follows is an intuitive explanation of semaphores, not a formal definition: A semaphore contains a non-negative integer variable, usually called its level. The two important operations on semaphores are up and down, which increment and decrement the level, respectively. However, when a call to down would decrement the level below zero, it blocks until a call to up is made (by another thread) that increments the level above zero. This is done in such a way that the level will never actually go negative. You could also say that the total number of completed down calls for a particular semaphore will never exceed the total number of up calls (not necessarily completed), plus its initial level.

sema_create
: A semaphore must be initialized to a certain level by calling this function. The initial level must not be negative.

sema_down
: Sema down operation. If count value is zero, block current thread untill a sema up operation was performed (by another thread).

sema_trydown
: Sema try down operation. If count value is zero, do nothing and return false, else decrement the semaphore counter and return true.

sema_up
: Increment a semaphore counter. If currently zero, and threads waiting for this sema, wakeup them in FIFO order. This case doesn't increment the sema counter !

PROGRAMMING INTERFACE

type semaphore = <abstract>

[semaphore] = **sema_create** ~level : int

[unit] = **sema_down** semaphore

[bool] = **sema_trydown** semaphore

[unit] = **sema_up** semaphore

VAM runtime environment

The VAM system consists of the virtual machine and the bytecode system with the already shown modules. These parts forming the base of the distributed operating system. But, there are many standalone programs building a full distributed operating system environment, like file and directory servers, teh boot server used for starting an initial VAM/Amoeba environment, several util programs and many more.

With the AMUNIX and VAM system, only a few programs are needed to build up a minimal Amoeba system. This minimal system consists of the following servers and utils:

flipd
: The Unix version of the Amoeba protocol stack. This is the core communcication interface used for loacl and remot RPC communication between clients and servers.

afs_unix
: A reimplementation of the Amoeba file server (Unix version), entirely written in OCaML. The filesystem is currently stored in two generic Unix files (one for the inode table, one for the filedata).

dns_unix
: A reimplementation of the Amoeba directory and name server (Unix version). The directory contents are stored in AFS files. The inode informations are currently stored in a generic Unix file.

vash
: The Amoeba-Unix shell. With vash it's possible to acces both, the Amoeba and the Unix file system (the last simply with a */unix* prefix), copying files, listing and managing of directories, performing standard requests (*std_info, std_status, std_destroy,...*), starting Amoeba processes (currently only on native Amoeba machines running with an Amoeba kernel), and many more.

Amoeba Hosts
: Of course, but optionally, several machines in the local network running a native Amoeba kernel.

Here the steps to build this minimal system (read the AMCROSS manual, too):

1. Create the flip administartion directory and the ethernet configuration file. Finally start the *flipd* program:

```
mkdir /amoeba/flip

vi /amoeba/flip/flip.conf
<<eth0=1:2a:33:42:51:66

flipd
FLIP: Configuration data:  eth0 (0:50:fc:1e:39:20)
FLIP: bpf ethernet if 0 has mac address:  0:50:fc:1e:39:20
   Ethernet interface 0:  bpf #0 has address 0:50:fc:1e:39:20
FLIP: process 1 connected.
FLIP: Debug interface started...
FLIP: Fast Local Internet Protocol Switch:
   AMUNIX Version 1.31 BSSLAB Stefan Bosse (sci@bsslab.de).
   (Sep 16 2004) Ready.
```

2. Extend the path settings in the local or global shell profile file:

vi /etc/profile
<< export PATH=$PATH:/amoeba/Vam-<version>/bin

3. Create the Amoeba file system (don't forget to create the directory specified by the -P option where the inode and data files will be placed by the server):

```
mkdir /amoeba/afs

afs_unix -c -o -n 100000 -P /amoeba/afs
>>
AFS: Creating Amoeba filesystem...
AFS: Blocksize:     512 [bytes]
AFS: Number of total blocks:     100000
AFS: Number of total inodes:     10000
AFS: inode part → /amoeba/afs/ldisk:00
AFS: data part    → /amoeba/afs/ldisk:01
AFS: Writing partition magic headers...  Done.
AFS: Writing super structure...  Done.
AFS: Resizing data partition...  Done.
AFS: Writing inodes...
AFS: Writing live table...     Finished.
AFS: Status → status ok
```

4. Start the AFS server:

```
afs_unix -s -P /amoeba/afs
>>
AFS: Atomic Filesystem Server, Ver.  1.06
   Stefan Bosse, BSSLAB (c) 2003
AFS: Initializing...
AFS: Opening partitions...
AFS: inode part → /amoeba/afs/ldisk:00
AFS: data part    → /amoeba/afs/ldisk:01
AFS: Reading the Superblock...  Done.
AFS: Checking the magic Header...  OK.
AFS: Label = "Filesystem"
AFS: Maximal number of files (inodes) = 10000
AFS: Blocksize = 512 bytes
AFS: Total number of blocks = 50000
AFS: Filesystem size = 25600000 bytes
AFS: Reading the livetime table...  Ok.
AFS: Creating data and inode caches...
AFS: Inode Cache →    1000 buffers of size    512 bytes
AFS: Data Cache    →    100 buffers of size    15360 bytes
AFS: Reading the Inode Table...
AFS: Found 0 used Inode(s).  A
FS: Found 0 valid file(s).
AFS: Found 0 free hole(s).
AFS: Biggest hole:  4361728 bytes.
AFS: Total free space:  21808640 bytes.
AFS: cache_om thread started...
```

```
AFS: starting 4 server threads...
FLIP: process 474 connected.
AFS: Ready.
```

Note: The default Path setting is: /amoeba/afs ; therefore the -P argument is not needed in this case.

5. Create the directory tree system:

```
mkdir /amoeba/dns
dns_unix -c -o -P /amoeba/dns -f1 /amoeba/afs/.servercap
>>
DNS: file server 1 capability:  6f:ff:ef:7f:cb:e9/0(ff)/45:4a:ab:e3:7e:67
DNS: file server 2 capability:  0:0:0:0:0:0/0(0)/0:0:0:0:0:0
DNS: Creating DNS tree...
DNS: Blocksize:     512 [bytes]
DNS: Number of total inodes (dirs):     10000
DNS: Writing partition magic headers...  Done.
DNS: Writing super structure...  Done.
DNS: Writing inode table...
DNS: Writing live table...
DNS: Creating root directory...Ok.  Finished.
DNS: Status → status ok
```

mkdir /amoeba/dns
Note: The default Path setting is: /amoeba/dns; therefore the -P argument is not needed in this case.

6. Start the DNS server:

```
dns_unix -s -P /amoeba/dns
>>
DNS: Directory and Name Server, Ver.  1.07
   Stefan Bosse, BSSLAB (c) 2003
DNS: Initializing...
DNS: Opening partition...
DNS: Inode part → /amoeba/dns/ldisk:02
DNS: Reading the Superblock...  Done.
DNS: Checking the magic Header...  OK.
DNS: entering one copy mode.  FLIP: process 475 connected.
DNS: checking file servers...Ok.
DNS: Label = "Filesystem"
DNS: Maximal number of files (inodes) = 10000
DNS: Blocksize = 512 bytes
DNS: Reading the livetime table...  Ok.
DNS: Creating directory and inode caches...
DNS: Inode Cache →    100 buffers of size    512 bytes
DNS: Dir Cache    →    30 entries
DNS: Reading the Inode Table...
DNS: Found 0 used Inode(s).
```

```
DNS: starting 4 server threads...
DNS: Ready.
```

7. Add the root capability path of your new crated DNS system to your profile (the unix prefix indicates a Unix path - it must not exist on your UNIX system!):

```
export ROOTCAP=/unix/amoeba/dns/.rootcap
```

8. Start the vash shell:

```
vash
>>
VASH - The Unix-Amoeba Integrator, version 1.11
   BSSLAB Stefan Bosse (c) 2003
[/]    >>
```

9. Create some directories and store some capabilities in the DNS system:

```
[/]
>> mkdir server
[/]
>> mkdir hosts
[/]
>> cd server
[/server]
>> get AFS /unix/amoeba/afs/.servercap
[/server]
>> set AFS afs
[/server]
>> get DNS /unix/amoeba/dns/.servercap
[/server]
>> set DNS dns
[/server]
>> dir -i
-------------------------------------------------------------------
Name              Info
-------------------------------------------------------------------
afs    Atomic Filesystem Server:  Super Cap
dns    DNS server capability
[/server]
>> cd /hosts
[/hosts]
>> mkhost "0:12:23:33:44:66" amhost1
[/hosts]
>> cd amhost1
[/hosts]
>> dir -i
```

```
[/hosts]
>> exit
```

10. You can shutdown the system directly from the Unix system:

```
std exit /unix/dns/.servercap
std exit /unix/afs/.servercap
```

The lowest level of execution environment is build with the OCaML virtual machine called *vamrun*. This VM directly executes machine independent ByteCode generated from the ML-Compiler.
This VM provides an automatic memory management. ML programmers don't need to borrow about memory allocation and freeing. This is done by the so called garbage collector GC.
On default, all array, list and string accesses are boundary checked. On failure, an exception is raised. For the case, the programmer doesen't caught this execption, the program willterminate with an exception message. Additionally, a backtrace of the current bytecode program is printed to screen. To enable backtracing of bytecode, you can either specify the *-b* option for the VM, or set the UNIX environment variable:

```
OCAMLRUNPARAM=b
export OCAMLRUNPARAM
```

This is the Unix version of the Amoeba filesystem server AFS (formerly known under the name bullet server). This is really the most simple file ever created. The filesystem consists of a number of contigous files only refernced with an object number. No file names, and no directory informations are handled. This is the task of the DNS server.

The server knows currently four operation modes:

1. Creation of a new filesystem.
2. Serviceing an already created filesystem.
3. Show status informations about the filesystem.
4. Perform a defragmentation of the filesystem. All files with a filesize greater a threshold will moved up to the end of the filesystem, if possible.

-s
Start the filesystem (already created).

-P
Partition directory (Unix). Default value: /amoeba/afs

-C
Directory name to store the super capability. Default: /amoeba/afs

-Nd
Number of data cache buffers. Default: 100

-Sd
Size of each data buffer. Default: 30 blocks

-Ni
Number of inode cache buffers. Default: 1000 buffers

-Si
Size of each data buffer. Default: 1 block

-t
Number of service threads. Default: 4 threads

-c
Create a new filesystem.

-n
Number of blocks. Default: 10000 blocks

-b
Block size. Default: 512 bytes. Should be kept untouched!

-i
Number of inodes = maximal number of files. Default: 10000 inodes

-P
Partition directory (Unix). Default value: /amoeba/afs

-C
Directory name to store the super capability. Default: /amoeba/afs

-o
Overwrite an existing filesystem.

-X

Show the status of all blocks of the filesystem.

-L

show cluster list [graphical default]

-D

Try a defragmentation of the filesystem. All files with a filesize greater a threshold will moved up to the end of the filesystem, if possible.

-M

maximal filesize theshold. Default: 10 blocks

This is the Unix version of the Amoeba Directory and Name server DNS (formerly known under the name soap server). The directory data infromations are stored in generic AFS (Atomic File system) objects, therefore a running AFS server is needed. The inode table and administration informations are kept currently in a generic Unix file.
This server performs name to capability mapping stored in directory like hirarchical structures. Each directory holds named objects, for example files, other directories, server capabalities, and stores for each name a capability pair (initially only the first one is used).
The directory server needs an already running AFS server to save his directory objects as usual file objects.

The server knows currently two operation modes:

1. Creation of a new directory system.
2. Serviceing an already created directory and name mapping system.

-s
Start the directory system (already created).

-P
Partition directory (Unix). Default value: /amoeba/dns
-C
Directory name where the super and root capability can be found. Default: /amoeba/dns
-Nd
Number of directory cache buffers. Default: 30
-Ni
Number of inode cache buffers. Default: 100 buffers
-Si
Size of each data buffer. Default: 1 block
-t
Number of service threads. Default: 4 threads

-c
Create a new filesystem.

-b
Block size. Default: 512 bytes. Should be kept untouched!
-i
Number of inodes = maximal number of directories. Default: 10000 inodes
-P
Partition directory (Unix). Default value: /amoeba/dns
-C
Directory name where the super and root capability can be stored. Default: /amoeba/dns
-F1
File server 1 capability. [format: x:x:x:x:x:x/o(r)/x:x:x:x:x:x]
-F2
Optional file server 2 capability. [format: x:x:x:x:x:x/o(r)/x:x:x:x:x:x]
-f1
File server 1 capability path. [Unix file name].
-f2
Optional file server 2 capability path. [Unix file name]
-m
Server mode [1:one (default), 2:two copy mode].
-N
Column names. Default: [Owner Group Other]

-R

Column rights. Default: [0xff 0x4 0x2]

-o

Overwrite an existing directory system.

Simple program which implements an interface to the Amoeba standard commands. Either server or object capabilities can be retrieved from the local UNIX system directly reading the capability stored in a UNIX file or from the Amoeba filesystem (AFS/DNS). The Amoeba filesystem is specified with the UNIX environment variable `ROOTCAP`.

Usage:

```
std [options] <path1> [<path2>]
```

Path convention:

```
path:  prefixed with /unix → local UNIX filesystem, else Amoeba DNS
```

Currently supported standard commands:

info
: Print an info string of a server or object from a server specified either with it's Amoebaor the UNIX path. The string is returned by the *STD_INFO* command.

status
: Print status informations of a server or object from a server specified with it's Amoeba or UNIX path. The string is returned by the *STD_STATUS* command.

exit
: Send an exit command to the specified server to shutdown the server. Normally full rights are required to shutdown a server.

The *vash* program is the main Amoeba operation platform in the Unix environment to control the Amoeba system. It's a simple shell with various Amoeba standard operations implemented. This shell supplies operations on both, the Amoeba and the host Unix filesystem. It's possible to copy file from the Unix to the Amoeba environment and vice versa. Nearly all builtin command work properly on both systems.
Commonly, server capabilities are published in the Amoeba directory and name system, called DNS. Simply spoken, the DNS (Directory and Name) server performs name to capability mappings. File data is, in contrast, saved on the fileserver AFS (Atomic Filesystem). The fileserver maps data to capabilities.
Therefore, nearly all operations supplied by this shell, will lookup server or object capabilities by path names, for example */server/afs*.

The following list show the available builtin standard server commands known from the Amobea operating system. There are two ways to resolve the server capability: from the DNS server, or from a Unix file (the path must start with the */unix* prefix). In the latter case, the capability is stored in a generic Unix file.

Examples

```
std_info /server/afs
std_exit /unix/amoeba/afs/.servercap
```

std_info [-h] <path1> [<path2>,...]
: Standard Info Request. Displays an information string returned by the server specified with the path arguments. All server should response to this request.

std_status [-h] <path1> [<path2>,...]
: Returns the status information returned by the server. Most server will response to this request.

std_exit [-h] <path1> [<path2>,...]
: This request performs a shutdown of the specified server, if supported. Only allowed with the unrestricted server capability.

std_destroy [-h] <path1> [<path2>,...]
: Destroy the object specified with the path argument.

std_age [-h] <path1> [<path2>,...]
: Decrement the live time of all objects from the server specified with the path argument. Not all server support this operation. Only these saveing objects permanently, like the DNS or AFS server. Objects with zero livetime will be destroyed. Only allowed with the unrestricted server capability.

std_touch [-h] <path1> [<path2>,...]
: This is the inverse operation. The livetime of objects specified with the path arguments will be set to the maximal livetime value. The maximal livetime is an internal server setting.

std_params [-h] <serverpath> [-s <paramname>:<paramval>]
: This request modifies or show internal server settings.

There are serveral operations on directories and files. Always, objects from the Unix filesystem must start with the */unix* prefix.

Example

```
cp /unix/amoeba/build/amcross/kernel/workstation/kernel /kernel/ws
cd /kernel
dir -l
dir /unix/etc
```

cp [-h -o -f] <source> <target>
: This operation copies a source file to the target place. Currently, the target path must contain the final file name. The source or the target path can be within the Unix file system. The *-o* option allows the overriding of the target, and the *-f* destroys a previously existing target. The difference: the first option only change the directory entry, the second deletes the file object, too.

ls — dir [-h -l -i -r -c] <path1> [<path2>,...]
: Show the directory listing or the object information specified by the path argument(s). There are different displays controlled by the options: the *-l* option results in a long listing with the directory entry names and the creation time, the *-i* option shows additional object informations (retrieved by the **std_info** request), the *-r* option shows the column rights of the directory entries, and the *-c* option shows the capabilities of the directory entries.

mkd — mkdir [-h] <path1> [<path2>,...]
: Create a directory.

rm — del [-h -d -f] <path1> [<path2>,...]
: Remove a row entry from a directory. This is not compareable with the Unix rm command. The *-d* allows the removing of a directory or of a server capability not reachable currently. The *-f* option destroys the object mapped by the removed row entry (== Unix rm /etc/hosts) .

cd <path>
: Change the current working to directory to path. Also recognized: *- and ..*

c2a [-h] <path1> [<path2>,...]
: Prints the object capability specified by the path in a human friendly format.

a2c [-h] ”0:0:0:...CAP-format” <path>
: Appends a capability in string format to the specified path.

There is a simple way in vash to handle capabilities with environment variables, similar to those already known from the Unix shell, but with the different, that they hold capabilities, rather strings.

get [-h -c] <envname> <path or cap string>
This command reads a capability, either lookuped by a filesystem path, for example */server/dns*, or directly with a string representation of the form *0:0:0../1(ff)/1:1:..*, in an environment variable.
<environment variable> := <path cap> — <cap string>.
The capability string must be used together with the *-c* option. The filesystem can be either the Amoeba or the Unix filesystem.

put [-h] <envname> <path string>
This command writes a capability from an environment variable to the filesystem (path string).
<path cap> := <environment variable>. The filesystem can be either the Amoeba or the Unix filesystem.

print_env [-h] <envname1> [<envname2> ...]
This command prints the capability conent of an environment variable.

With vash, the user can start Amoeba programs on native Amoeba hosts. The binary file must be currently an AFS file.

ax [-h -v -E <earg>] <hostpath> <progpath> [-<progargs>,..]
First, a kernel capability must be specified (hostpath), for example */hosts/dio*, or simply the host name without a path (assuming the default host path */hosts*). Next, the binary file must be specified (progpath), for example */utils/io*. Finally, program arguments can be specified. Because at least a process wants to print informations on the standard output and error channel, there must be a terminal (TTY) server. Vash is provided with a simple TTY server (requires starting vash with the *-t* option to enable the server inside vash). The *TTY* environment is automatically set to the internal server capability. Otherwise, another TTY server can be specified (for example the one in the kernel */hosts/dio/tty:00*).

The *vax* util can be used to start native Amoeba programs directly from the UNIX environement. For this purpose, it supplies a terminal server TTY and a reduced file server AFS, mapping the program UNIX file to an AFS object. Additionally, some kind of process control is performed. Vax is also responsible to build up the stack segemtn for the native Amoeba process (needed for the kernel process server which loads the stack segement as a usual data segment from the AFS server!).
To start a native Amoeba program file located on the UNIX filesystem, the path preceeded with */unix* must be provided with the program imgae filename and the host name of the native Amoeba machine. This host name (and the corresponding object capability) must be present in the root Amoeba filesystem (commonly the AFS/DNS UNIX filesystem used within the VAM system). The default lookup path for the host capability is */hosts*.
The recently started Amoeba process needs usually a minimal capability environment consiting of the Termnial- (*TTY*) , Random- (*RANDOM*) and Time Server (*TOD*) capability. The first is provided by vax, the latters commonly by the kernel. The last two are resolved automatically by vax. The capability can be set/overriden with the *-E* option.
Also VAM bytecode programs can be started. In this case, the virtual machine (for the nativa Amoeba target - VAMRAW) must be specified. Either using the *-vm* option or the environment variable *VAMRUN*.
Path convention for binaries:

/unix/...
: located on local UNIX filesystem,

/..
: located on the root AMoeba filesystem, specified with the *ROOTCAP* environment variable.

```
vax [-h -v -k] [-E NAME=PATH] [-S NAME=STR] [-vm <vmpath>]
   <host> <prog> <prog arguments>
```

-E
Add environment capability name <*capname*> lookuped from <*path*>. NAME=% resolves the default capability from the kernel host directory (only standard caps: TOD,RANDOM,TTY,PROC,...).

-S
Add string environment variable (the same type as with UNIX).

-k
Boot a new kernel image on given host machine.

-vm
Path to the target system vamrun machine. Alternatively use the VAMRUN environment variable to specify it.

<**host**>
Either absolute kernel host root path or host name, found in the Amoeba filesystem: Default host path: /*hosts*

<**progpath**>
UNIX or Amoeba path and filename of the native Amoeba or bytecode program binary.

```
vax -E TTY=% juki01
   /amoeba/Amcross/driver/i86_pc/cnc /hosts/juki01
vax -k juki01 /amoeba/Amcross/kernel/i86_pc/workstation -noreboot:1
```

A virtual disk server for UNIX. This server provides an Amoeba virtual disk interface for a local UNIX device, like an harddisk or a compact flash memory card. Normally, this server resides within the Amoeba kernel and is coupled to the low level device drivers of permanent storage devices.
The virtual disk server is needed for the bootdisk server implementing a simple boot filesystem, for installing kernels, or for implementing an Amoeba filesystem (AFS and DNS) using a local UNIX device.

```
vdisk [options] <device>
```

-H

Specify the host path directory in the Amoeba filesystem directory to store the vdisk capabilities in. The default value is: `/hosts/<UNIX host name>`

-b

Blocksize in bytes. Default value: 512 Bytes.

-n

Number of service threads. Default value: 4 threads.

-chs #c,#h,#s

If the storage device contains already a valid Amoeba filesystem, the vdisk server is capable to retriev the (physical/logical) geometry informations of the device. If no so, teh user must specify the geometrical data: number of cylinders (c), number of heads (h), number of sectors (s).

-i

Print disklabel and partition informations of the disk.

-v

Verbose mode. Prints additional informations during run.

-h

Print a help message and exit.

The *xafs* program is a graphical file browser and copy util. It's devided in the Amoeba filesystem specified with the ROOTCAP environment variable) and the local UNIX filesystem. You can create and delete directories, copy file from UNIX to Amoeba and vice versa. Additionally, object status informations can be displayed.

Tutorial: ML-VAM programming

...

If a function calls serveral subfunctions, each returning a status value, and the next operation must only be performed if the previous status is *std_OK*, then OCaML's exception mechanism should be used.

Example

```
let demo () =
try
begin
   let stat = fun1 () in
   if (stat <> std_OK) then
   raise (Error stat);
   let stat = fun2 () in
   if (stat <> std_OK) then
   raise (Error stat);
   ...
end
with
   | Error err → err
```

The RPC server loop will call special functions depending on the command delivered within the request header of the RPC. Don't abuse the if statement to select the current RPC command. Instead use a modificated match statement:

Example

```
let stat,n,hdr_req = getreq (serverport,reqbuf,reqsize) in
if (stat = std_OK) then
begin
   match hdr_req.h_command with
   | com when com = std_INFO →
   begin
   ... serve the RPC ...
   end;
   | com when com = myCOM →
   begin
   ... serve the RPC ...
   end;
   | _ → ...
end;
ignore( putrep (rep_hdr,rep_buf,rep_size));
```

The ManDOC documentation tool

The ManDOC documentation system provides a programming language for the description of the content of software or other documentation pages, and is built upon the *mandoc.cma* library, implemented with ML like all other parts of the VAM system. The implementation provides a ManDOC frontend, capable of reading the document description language, similar to the ancient roff format, and several backends for formatted output of the document content, like LaTex, HTML and terminal text output for online help support. This document you're currently reading was written and formatted using the ManDOC tools.
Most ManDOC commands consist of a two character dot and antidot pair with mirrored command name, for example:

.S1
.NA `Section name` **.AN**
.1S

Each dot and antidot command must be surrounded by at least one space character, otherwise the command is ignored.
There is no strict document structure. But there are generic sections of different depth and a manual page section. Within a section, text lines build paragraph blocks, just like with LaTex. There are special blocks used beside text paragraphs, for example lists. Additionally, there is simple table support. Table 19 gives an overview about the ManDOC elements.

(TAB. 19) DOCUMENT STRCUTURE AND ELEMENTS

Body	**Sections**	**Special Blocks**	**Text styles**
Title	S1	Ordered Lists	Bold
	S2	Unordered Lists	Italics
	S3	Argument Lists	Typewriter
	S4	Literature Lists	Bold-Italics
	Manual Page	Figures	AsIs
	MP Paragraph	Tables	
		Dataformat Tables	
		Examples	
		Programming Interfaces	

One main feature of the ManDOC system is the support of programming interfaces. Only the content of an programming interface, like a function or structure, must be provided, not the alignment or layout of the single parts (names, arguments,...) of an interface. This feature speeds up documentation of programming manuals considerable.

A ManDOC document is divied in sections like with any other structured formatting systems. But there is no strict convention to handle the depth of sections and the order. You can usee the ManDOC system for formatting of simple manual pages and books, too.
Each section is started enclosed with a command pair, shown in table 20. The next command which follows the first section command is the name of the section, enclosed with the name command pair. Generic text paragraphs need no special environment, in contrast to latex or html. They are consist just of text elements.

(TAB. 20) MANDOC SECTIONS

Command pair	**Description**
`.TL <cont> .LT`	Title of the document (optional)
`.S1 <cont> .1S`	Generic Section 1
`.S2 <cont> .2S`	Generic Section 2
`.S3 <cont> .3S`	Generic Section 3
`.S4 <cont> .4S`	Generic Section 4
`.MP <cont> .PM`	Manual Page
`.MA <cont> .AM`	Manual Page Paragraph
`.NA <text> .AN`	Title name of a section

A manual page environment consists of several manual page paragraphs with a paragraph name. The manual page has a hierarchy depth comparable with the S3 section, and each manual page paragraph is like the S4 section.

EXAMPLE *Sections*

```
.S1
    .NA The first section .AN
    Here comes text of the first paragraph.
    And another section.
    .S2
    .NA A subsection .AN
    More text...
    .2S
.1S
```

Within sections or generic paragraphs there are some special blocks, for example for including of images. Images can be optionally included in figure environments with an additional text header. Other common environments like lists are supported, too.

Figures

An adjusted figure environemnt with an image and a explanation text paragraph. Includes the image environment and the text line. The generic format of a figure environment:

```
.FG
    .PI
    .NA <image name> .AN
    .IP
    <figure description paragraph>
.GF
```

Tables

There is limited support of tables inside the ManDOC system. Tables consist of rows and columns. The print layout of the table depends on the used backend. The generic format of a table environement:

```
.TB
    .TR
    .TC <row 1 col 1> .CT
    .TC <row 1 col 2> .CT
    .RT
    .TR
    .TC <row 2 col 1> .CT
    .TC <row 2 col 2> .CT
    .RT
.BT
```

Data Format Tables

Can be used for formatted data structure representations. Enables the usage of column groups. The cell width in character units must be specified in the necessary header. Each data column entry must specify the cell width in cell units (1,2,3...). A number greater than 1 means a column group. Generic format:

```
.DF
    .TH
    .DE .VE <width in char units col 1> .EV .ED
    .DE .VE <width in char units col 2> .EV .ED
    .HT
    .DR
    .DE .VE <width in cell units> .EV <text> .ED
    .DE .VE <width in cell untis> .EV <text> .ED
```

.RD
.FD

Examples

A special example environment with an optional title name exists. The text in this environment will be displayed as is with indention using spaces.

.EX
.NA `<title text>` **.AN**
`<preformatted text lines>`
.XE

Lists

There are four types of lists: ordered, unordered, argument and literature lists. Each list consists of an list environment and list items:

.OL
.LI `<item1>` **.IL**
.LI `<item2>` **.IL**
.LO

The argument and reference lists have named list items (the first part of these list items must be a name specifier).

.AL
.LI .NA `<name>` **.AN** `<item1>` **.IL**
.LI .NA `<name>` **.AN** `<item2>` **.IL**
.LA

Lists can be nested, with same or different kinds of lists.

Example

The Example paragraph provides preformatted text, for example source code. The title name is optional. In contrast to generic text paragraphs, the preformatted text includes white spaces! There is an example at the end of this section.

.EX
.NA `Pipes` **.AN**
`# echo "" | file`
.XE

Html

Native HTML source can be included affecting only the HTML backend.

Tex

Native LaTex source can be included affecting only the LaTex backend.

(TAB. 21) MANDOC FIGURES AND EXAMPLE	
Command pair	**Description**
`.FG <cont> .GF`	A figure environemnt (with image)
`.PI <cont> .IP`	An image name. Optional included in an figure environment.
`.EX <cont> .XA`	An example environment.

(TAB. 22) MANDOC GENERIC TABLES	
Command pair	**Description**
`.TB <cont> .BT`	A simple table environment.
`.TH <row> .HT`	Table header (title text).
`.TR <row> .RT`	One row of a table.
`.TC <col> .CT`	One date entry (column) of a table row.

(TAB. 23) MANDOC DATAFORMAT TABLES	
Command pair	**Description**
`.DF <cont> .FD`	A simple table environment.
`.DH <row> .HD`	Table header specifies cell widths.
`.DR <row> .RD`	One content row of this table.
`.DE <col> .ED`	One date entry (column) of a table row.
`.VE <int val> .EV`	Cell width value.

(TAB. 24) MANDOC LISTS	
Command pair	**Description**
`.OL <cont> .LO`	An ordered list.
`.UL <cont> .LU`	The same as an unordered list.
`.AL <cont> .LA`	An argument item list.
`.RL <cont> .LR`	A litertaure/reference list.
`.LI <cont> .IL`	One list item of a list.

(TAB. 25) MANDOC SPECIAL SOURCE	
Command pair	**Description**
`.HS <cont> .SH`	Html source.
`.TS <cont> .ST`	LaTex source.

```
.S1
   .NA The first section .AN
   Here comes text of the first paragraph.
   And another section.
   .S2
   .NA A subsection .AN
   Now a figure follows.
   .FG
   .PI .NA vam_am_detail.eps .AN .IP
   (Fig.1) All the components of the VAMNET system together
   .GF
   Now we have an example block:
   .EX
   .NA My example .AN
int c;
int fun(){
   int a=0;
   return a;
};

   .XE
   An ordered list:
   .OL
   .LI Here comes the first list item .IL
   .LI and the next one ...  .IL
   .OL
   But an argument list has this format:
   .AL
   .LI .NA Arg1 .AN the first named list item .IL
   .LI .NA Sec2 .AN and the second one.  .IL
   .LA
   .2S
.1S
```

Here is an example for native HTML source code included in the ManDOC document:

```
.HS
<P>
More informations ...
<IMG src="info.jpg">
</P>
.SH
```

and the result (only visible in HTML output):

Within paragraphs there are only text and some special elements, for example a forced paragraph break. Some simple font specifiers exist: bold, italic, bolditalic and typewriter text.
Special symbols, for example greek letters, can be included with their latex names. See latex documentation for details. The printed output can depend on the backend file format. Html or generic text output have only limited support of special symbols. It's possible to make a page link to another section. For this case, the referred section name must immediately followed by a label name. The reference mechanism depends on the used backend and can be limited or non existing.

(TAB. 26) MANDOC SPECIAL PARAGRAPH ELEMENTS

Command pair	**Description**
`.SX` <`text`> `.XS`	A special symbol (with latex names).
`.RF` <`text`> `.FR`	Page reference to a section with a given label name.
`.LB` <`text`> `.BL`	The label name of a section.
`.NL`	Paragraph break.
`.VE` <`text`> `.EV`	A special value. Needed for some command environments.
`.B` <`text`> `.R`	Bold font style text.
`.I` <`text`> `.R`	Italics font style text.
`.BI` <`text`> `.R`	Bold Italics font style text.
`.T` <`text`> `.R`	Typewriter font style text.
`.[` <`text`>.]	Keep text as is (typewriter style).
`.SB` <`text`> `.BS`	Subscript style.
`.SP` <`text`> `.PS`	Superscript style.

EXAMPLE *Paragraph elements*

```
.S1
    .NA The first section .AN
    .LB S1013 .BL
    Here comes .B bold .R text of the first paragraph.  .NL
    Here starts a new paragraph.  And another section.
    .S2
    .NA A subsection .AN
    See section 1 .RF S1013 .FR for details.  Sometimes the
    greek symbol .SX alpha .XS is needed!
    .2S
.1S
```

One main advantage of the ManDOC system is the support of preformatted programming interfaces. Only the content of a function, data type or data structure must be specified. The user must not be concerned about the proper layout of this programming interface. This is done by the ManDOC backends. Supported languages are ML (with class and object support) and C.
Programming interface entries are collected in an interface environment. All programming interfaces expect a name specifier and some arguments. Functions expect at least one return argument (ML: multiple return arguments means a return tuple). Argument can be followed by a comment.

Functions

Generic format (here: ML, C allows only one return argument - of course):

```
.IF
    .NA <function name> .AN
    .RV <ret 1> .VR
    .RV <ret 2> .VR
    .AR <arg 1> .RA
    .AR <arg 2> .RA
.FI
```

A ML function argument can be also in the uncurried form (data tuple):

```
.IF
    .NA <function name> .AN
    .RV <ret 1> .VR
    .RV <ret 2> .VR
    .AV <arg 1> .VA
    .AV <arg 2> .VA
.FI
```

Types and structures

Generic format:

```
.IT
    .NA <type name> .AN
    .AV <type 1> .VA
    .AV <type 2> .VA
.TI
```

(TAB. 27) MANDOC PROGRAMMING INTERFACES	
Command pair	**Description**
`.IN <content> .NI`	Programming interface environment.
`.IF <cont> .FI`	ML Function
`.IV <cont> .VI`	ML Value
`.IE <cont> .EI`	ML external value
`.IT <cont> .TI`	ML type list
`.IS <cont> .SI`	ML structure
`.IX <cont> .XI`	ML exception
`.IM <cont> .MI`	ML module
`.MC <cont> .CM`	ML object class
`.MT <cont> .TM`	ML class method
`.OB <cont> .BO`	ML class object
`.CF <cont> .FC`	C function
`.CV <cont> .VC`	C variable
`.CY <cont> .YC`	C type defintion
`.CS <cont> .SD`	C structure
`.CH <cont> .HC`	C Header

(TAB. 28) MANDOC PROGRAMMING INTERFACE ARGUMENTS	
Command pair	**Description**
`.AR <text> .RA`	Curried argument (functions, types, ... : C,ML)
`.AV <text> .VA`	Uncurried argument (tuple: only ML)
`.RV <text> .VR`	Fucntion return value (C,ML)
`.MU`	Mutable structure entry (within argument, ML)
`.PV`	Private class entry (ML)
`.VT`	Virtual class entry (ML)
`.(* .*)`	Argument comment (within argument: C,ML)

EXAMPLE *Programming Interface*

.IN
  .NA `ML and C` **.AN**
  .IF
  .RV `port` **.VR**
  .NA `ml_port_new` **.AN**
  .AR `()` **.RA**

.FI
.IF
.RV int .VR
.NA c_port_new .AN
.AR () .RA
.FI
.NI

The ManDOC system is entirely implemented with ML. There are the ftonend module *doc_core*, and several backends for various output formats: *doc_latex* for latex, *doc_html* for html and *doc_txt* for ascii text (terminal) output.
First, all the ManDOC input text must be devivded in atoms simply by breaking all the text in a space separated token list, and then parsed and converted to an internal structure represenation. This is done with the *atoms_of_file* and the *tree_of_atoms* functions, respectively. The prepared internal structure of the dcoument is finally passed to the desired backend function, for example *tex_of_tree* to produce formatted output. The result is written into an file.

The backends functions can be controlled with several options, at least the output file name is specified with an option ([H]: Html, [L]: Latex).

PROGRAMMING INTERFACE *Options*

```
type doc_options =   Doc_single          ▷ One single output file [H] ◁
                   | Doc_multi_s1        ▷ One file for each new section S1 [H], Pagebreak [L] ◁
                   | Doc_multi_s2        ▷ One for each section S2 [H], Pagebreak [L] ◁
                   | Doc_multi_s3
                   | Doc_multi_s4
                   | Doc_multi_mp
                   | Doc_with_toc        ▷ Print a table of content ◁
                   | Doc_link_ref        ▷ Linked references ◁
                   | Doc_Main of string  ▷ Main filename without extension ◁
                   | Doc_pdftex          ▷ Latex with pdf target ◁
                   | Doc_color           ▷ Colored output ◁
```

The public interface consists of the following functions. First the input text (either from a string or read from a file) is split into atoms, and followed by the mean parser function.

atoms_of_text
: Returns a list of all lines from 'text', and each list member is a string list of text atoms [delimited by spaces].

atoms_of_file
: Reads the text from a file and convert it to an atoms list.

tree_of_atoms
: Builds a structure tree from the atom list of the text.

PROGRAMMING INTERFACE *Basic types*

```
type structure_attr =   T_Bold
                      | T_Italic
                      | T_BoldItalic
                      | T_Type
                      | T_AsIs
                      | T_Subscript
                      | T_Superscript

type structure_content =   S_Empty
```

```
| S_Text of string
| S_Body
| S_Title
| S_TOC
| S_S1
| S_S2
| S_S3
| S_S4
| S_MP
| S_MP_paragr
| S_Figure
| S_Table
| S_DataFormat
| S_Interface
| S_Example
| S_Preform
| S_ML_Fun_Interface
| S_ML_Ext_Interface
| S_ML_Type_Interface
| S_ML_Exc_Interface
| S_ML_Struc_Interface
| S_ML_Method_Interface
| S_ML_Object_Interface
| S_ML_Module_Interface
| S_ML_Class_Interface
| S_C_Hdr_Interface
| S_C_Fun_Interface
| S_C_Var_Interface
| S_C_Type_Interface
| S_C_Struc_Interface
| S_Attribute
| S_OList
| S_UList
| S_ArgList
| S_LitList
| S_List_Item
| S_TableHead
| S_TableRow
| S_TableCol
| S_TableNoRulers
| S_DataHead
| S_DataRow
| S_DataEntry
| S_Value
| S_Name
| S_Mutable
| S_Private
| S_Virtual
| S_CurArg
| S_UnCurArg
| S_RetArg
| S_NL
| S_TAB
| S_Link
```

| S_Ref
| S_Label
| S_Comment
| S_Image
| S_Symbol

type structure_block = { **mutable** s_parent: structure_block option ;
mutable s_childs: structure_block list ;
mutable s_content: structure_content ;
mutable s_attr: structure_attr list ;
mutable s_line: int ;
mutable s_name: string ref }

type section_names = Sec_s1 of string
| Sec_s2 of string
| Sec_s3 of string
| Sec_s4 of string
| Sec_mp of string

PROGRAMMING INTERFACE *Functions*

[*atoms*: string list] = **atoms_of_text** ~text : string

[*atoms*: string list] = **atoms_of_file** ~fname : string

[*tree*: structure_block] = **tree_of_atoms** ~atoms : string list

[unit] = **print_tree** *tree*: structure_block

The latex backend creates a tex source file which must be translated with the *latex* typesetting system into the device independent interchange format dvi, and finally with the *dvips* program into postscript. Alternatetively, the output can be perpared for the *pdflatex* system to produce pdf output instead dvi. The tex or pdf output filename must be set with the *Doc_Main* option. An optional section list can be specified for the case, only a part of a document (a subsection for example) should be formatted, and the sub document context must be specified.
The *Doc_color* option may not be used together with the *Doc_pdflatex* option due to a bug in the *pdflatex* system!

PROGRAMMING INTERFACE

[unit] = **tex_of_tree** ~ds : structure_block →
~options : cod_options list →
~sections : section_names list

The html backend creates either one (huge) html file or several html files breaked by section boundaries, specfied with the *Doc_multi_sX* options.
The main output filename must be set with the *Doc_Main* option. An optional section list can be specified for the case, only a part of a document (a subsection for example) should be formatted, and the sub document context must be specified.
Currently, this backend produces only HTML 4 transitional output without CSS support. In the case of a multifile output, there are navigation bars on the beginning of each file (content up and index link), and at the end for the next following or the previous section of the same section level, if any.

PROGRAMMING INTERFACE

[unit] = **html_of_tree** ~ds : structure_block →
~options : cod_options list →
~sections : section_names list

Debugging

There are several ways to get informations about the state and to manipulate the state of VAM processes:

➤ The *Db* Module: simply prints information to the standard output channel of the process depending of a debug level.

➤ The inline debugger using the *Debugger* module. Using the Amoeba RPC system, external programs can connect to this simple debugger. For example the current thread states with stack traces can be requested.

Each VAM-ML program can start an internal builtin debugger for both getting state informations and setting for example thread states, variable contents and so on. Currently only information management is implemented. To use this debugger, all modules MUST be compiled and linked with the additional debug information flag '-g'!
The debugger is located in the ML-Module *Debugger* from the server library. A program being able to debug must call the *Debugger.init debugger* function to start a server thread waiting for requests on a server port generated from a string of size 8. This is the **private port** of the server. The port name, only consisting of valid characters 'a'-'z','A'-'Z','0'-'9', randomly generated, is printed on the standard out channel of the process. Using this portname, the internal debugger thread can be accessed from any other process in the VAM/Amoeba environment.
The currently most powerfull client function is the *Debugger.debug_trace* function. This function prints the current thread states and the stack trace of the specified process.
The following example shows the usage of this debugger interface. To get information about the current thread states of a VAM process, simply call the debug_trace function within the vam toplevel system with the published port name.
The internal debugger of the target process will examine the current stack of each thread and tries to find a valid **entry module**, that means a ML function currently calling an external C function which blocks this thread, for example a thread waiting for a locked mutex. If there is such an entry function (marked with the key word **Entry Module**), the stack is iterated up to the stack top (using the stacksize debug informations supplied from the compiler), or untill a non resolved debug event was found, that means a function address without any debug informations, like module name, source location and other informations. All found and recognized function frames with their source module name, line and char position within the source file, are printed.
If there is no information about the current ML function, the debugger tries to find a valid function code address on the stack. On (maybe doubtfull) success, this function is marked with the key word **Find Module**. From this new starting point within the stack, the stack is iterated upto the top (using again the stack size debug informations supplied from the compiler) to find more function frames.
Other informations in the thread trace are the VM registers PC and SP, the program counter and the stack pointer respectively. Together with informations from the kernel about threads of a process, it's possible to solve many thread related runtime problems, for example dead locks. The following example gives an impression of such a problem.

EXAMPLE *VM thread debugging*

```
The AFS server was started with the '-d' option, which starts the debugger
thread:

% vax /hosts/geo01 afs -s -d
DEBUGGER: Portname="S9oXQMko"
DEBUGGER: service thread started...
AFS: Atomic Filesystem Server, Ver.  1.09
   (C) 2003 BSSLAB Stefan Bosse
AFS: Initializing...
AFS: Opening partitions...

Now, from another console using vam, the current thread states can be
requested simply typing:
```

```
% vam

===========              VAM system              ===========
          [Version 1.7 (Build date Jan    6 2005)]
          Written by Stefan Bosse (sci@bsslab.de)
          (c) 2003 by BSSLAB

   Use 'help "help"' or 'help "intro"' to get more informations.
   Loading initial environment...
   Loading online help system...
   Ready.
[] Debugger.debug_trace "S9oXQMko" ;;
Current Thread 1 >>>>
   STACK: HIGH=0x400ac050 LOW=0x400a4050
   REG: SP=0x400ac00c PC=0x40142cdc
   Entry Module Debugger    Line=    101    Pos= 24 Stacksize= 10

Other Thread 0 >>>>
   STACK: HIGH=0x400bde00 LOW=0x400b5e00
   REG: SP=0x400bdd60 PC=0x4014ed14
   Entry Module    Main    Line=    304    Pos= 27    Stacksize=    9
   Module        Main    Line=    491    Pos= 16    Stacksize= 27

Other Thread 6 >>>>
   STACK: HIGH=0x40431da0 LOW=0x40429da0
   REG: SP=0x40431c54 PC=0x401292c8
   Find Module    Afs_cache    Line=    816    Pos= 26    Stacksize=    5
   Module        Afs_vdisk    Line= 1598    Pos= 55    Stacksize=    3
   Module    Afs_server    Line=    831    Pos= 66    Stacksize= 17
   Module    Afs_server_rpc    Line=    234    Pos= 65    Stacksize= 32
   Module        Thread    Line=    90    Pos= 14    Stacksize=    5

Other Thread 5 >>>>
   STACK: HIGH=0x4041b4a0 LOW=0x404134a0
   REG: SP=0x4041b35c PC=0x401292c8
   Find Module    Afs_vdisk    Line= 1786    Pos= 38    Stacksize=    5
   Module    Afs_server    Line=    334    Pos= 54    Stacksize=    7
   Module    Afs_server    Line=    655    Pos= 58    Stacksize= 13
   Module    Afs_server_rpc    Line=    205    Pos= 54    Stacksize= 30
   Module        Thread    Line=    90    Pos= 14    Stacksize=    5

Other Thread 4>>>>
   STACK: HIGH=0x403b3af0 LOW=0x403abaf0
   REG: SP=0x403b3a64 PC=0x4013cce4
   Entry Module Afs_server_rpc    Line=    123    Pos= 70    Stacksize= 22
   Module            Thread    Line=    90    Pos= 14    Stacksize=    5

Other Thread 3 >>>>
   STACK: HIGH=0x4039d0b0 LOW=0x403950b0
   REG: SP=0x4039d024 PC=0x4013cce4
   Entry Module Afs_server_rpc    Line=    123    Pos= 70    Stacksize= 22
   Module            Thread    Line=    90    Pos= 14 Stacksize=    5
```

```
Other Thread 2 >>>>
   STACK: HIGH=0x40394110 LOW=0x4038c110
   REG: SP=0x4039
4018 PC=0x401292c8
   Find Module    Afs_vdisk    Line= 1118    Pos= 38    Stacksize=    1
   Module       Afs_cache    Line= 1980    Pos= 44    Stacksize=    8
   Module       Afs_cache    Line= 2005    Pos= 49    Stacksize=    2
   Module          Hashtbl    Line=    141    Pos= 13    Stacksize=    4
   Module          Hashtbl    Line=    144    Pos= 19    Stacksize=    6
   Module       Afs_cache    Line= 2015    Pos= 25    Stacksize=    5
   Module       Afs_vdisk    Line= 2245    Pos= 55    Stacksize=    1
   Module          Thread    Line=    90    Pos= 14    Stacksize=    5

%vash
>> kstat -m /hosts/geo01
...
Process 1:
prio=1 nthr=7 nrun=0 flags=0
Tid Pri Sys    Event    Mutex    Timeout    StkBot    StkTop    Flags
57    11    27       3d27fb0S       3d29000    3d31000       0
55    11    27       3b56100          3988000    3990000       0
54    11    27       3b560d0          39a0000    39a8000       0
58    11    -1    290110               39b8000    39c0000       0
56    11    -1    28ffe8               39d1000    39d9000       0
53    11    27       3b560d0          39e9000    39f1000       0
42    11    -1    28f7d0               3c01000    3c09000       0
...
```

There are several ways to get informations about the state of the kernel:

- Kernel statistics accessible with the *KSTAT* request from the kernel system server if compiled with *STATISTICS* enabled:

```
0    dump 3c59x statistics
3    dump 3c9xx statistics
9    dump 3c509 statistics
A    dump event info
C    flip rpc dump
E    Ethernet flow control statistics
F    flip routing table
G    flip group dump
I    flip interface
M    dump mutex info
N    flip network dump
P    packet pool usage
R    dump rtl8129/8139 statistics
S    dump software timer info
X    IPC statistics
Y    dump ndp statistics
b    dump random seed bit info
c    flip rpc statistics
d    dump ei8390 statistics
e    dump Ethernet statistics
f    flip fragmentation dump
g    group statistics
i    flip interface statistics
k    flip rpc kid dump
l    dump Lance statistics
m    dump thread table
n    flip network statistics
p    flip rpc port dump
r    dump hardware timer statistics
s    dump segtab
t    dump all hardware resources
u    print uptime
v    print version
w    raw flip interface dump
x    dump I/O ports
y    dump I/O memory
z    dump IRQ list
```

- Kernel tracing accessible from the kernel system server if compiled with *TRACING* enabled:

  1. Thread trace
  2. Ethernet packet trace

3. FLIP message trace

References

[KAS93]
M.F. Kaashoek, R. van Renesse, H. van Staveren, and A.S. Tanenbaum *FLIP: an Internetwork Protocol for Supporting Distributed Systems* ACM Transactions on Computer Systems, pp. 73-106, Feb. 1993.

[AMSYS]
Amoeba 5.3 System Administration Guide

[AMPRO]
Amoeba 5.3 Programming Guide

[COU98]
G. Cousineau, M. Mauny `The Functional Approach to Programming` Cambridge Press, 1998

[Fab99]
Software: Fabrice Le Fessant, projet Para/SOR, INRIA Rocquencourt.

[OCAML305]
Software: OCaML version 3.05, Xavier Leroy et al., projet Cristal, INRIA Rocquencourt