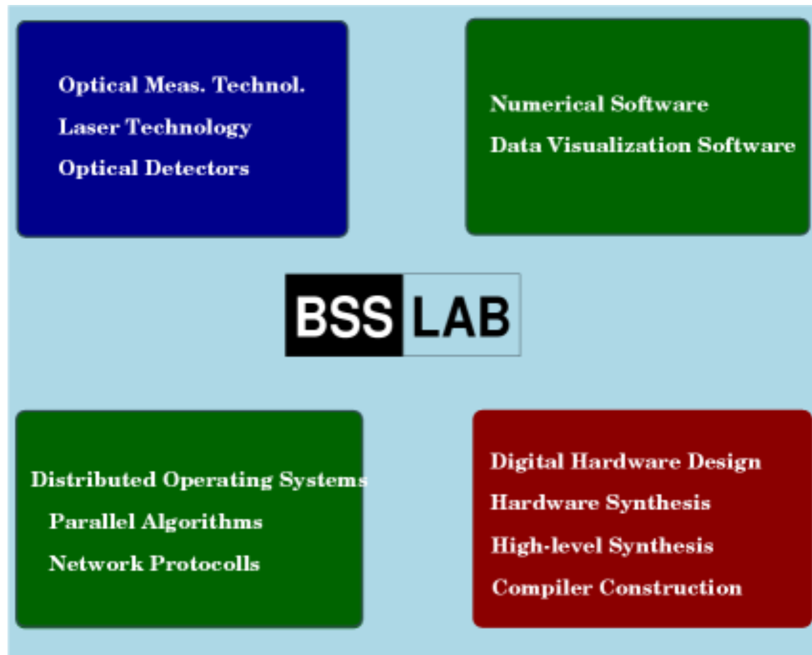
WWW.BSSLAB.DE



Independent Research and Development Laboratory

Scientific Measuring and System Techniques

BSSLAB – Dr. Stefan Bosse
Vohnen Street 86
D–28201 BREMEN

Germany

Phone: (49)421/21864103 or (49)421/875215
Fax: (49)421/875215

VX−Kernel

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.

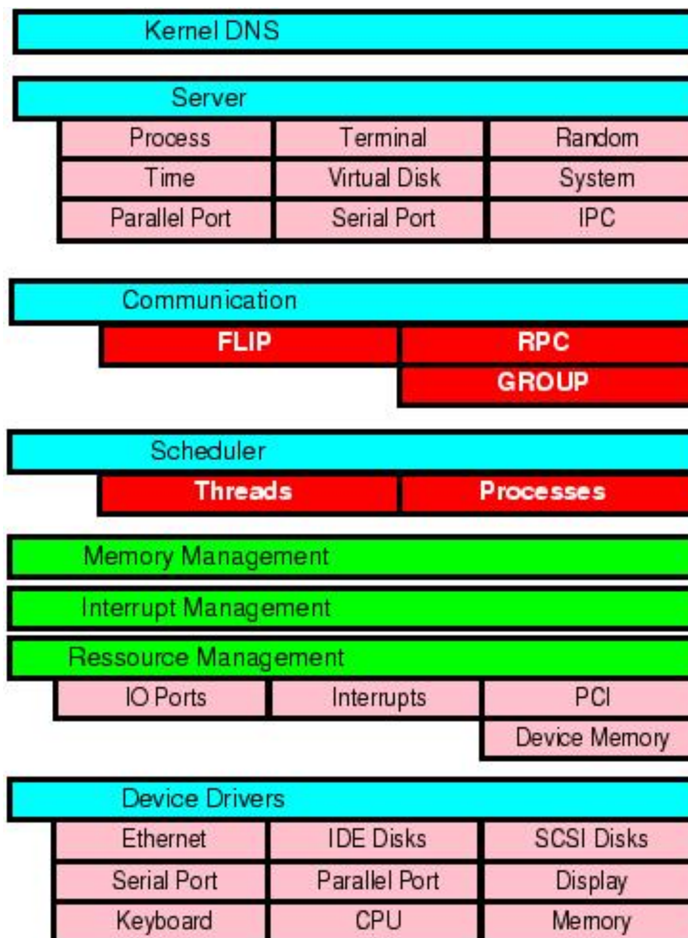
The kernel has the following features and avtanges:

- 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 1 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. 1) 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 publishs 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.

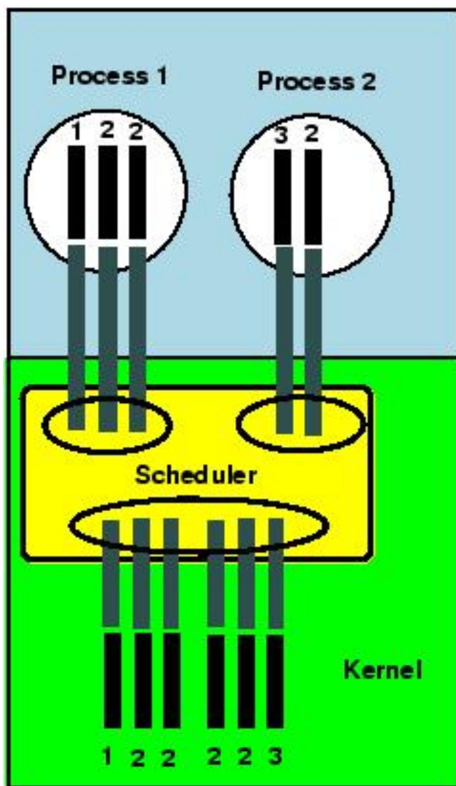
Another important part of the kernel is the **time(r) management**. In contrast to traditional kernels has the VX−Kernel a dynamically timer management, that means there is no fixed time unit (ticks). The two jobs of the kernel timer management are:

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. 2) 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}

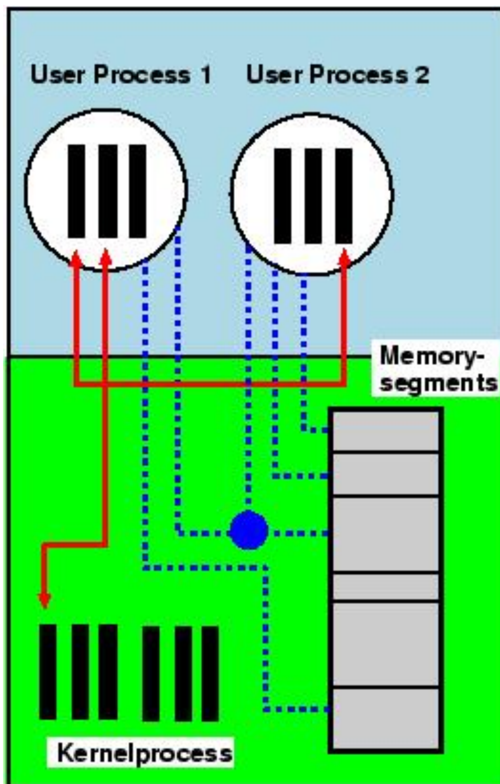
   The process priority has a higher weight than the thread priority.

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 3 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. 3) 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.

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

---

Generated by MANDoc (C) 2005 BSSLAB Dr. Stefan Bosse
Revision 1113144380

(C) BSSLAB Dr. Stefan Bosse, Revision 1138111946