# The Borium kernel concept

x86-micro-kernel | v0.0.5

doc version 0.2

## Content

## Development Team

**Project Leader / Project Admin / Main Developer (Kernel)**

Harry Hornbacher - mizztajonny

**Project Admin / Main Developer (Kernel - LibCpp)**

George Karpouzas - gkarpouzas

## Kernel overview

The Borium kernel will be an implementation of a micro-kernel. Inspired by many other open source projects[i].

The tasks of the kernel are to provide

- ➔ *Resource Interfaces*

  The ResourceManager controls the lowlevel access for user programs.

  - ➔ *Hardware Managers* (CPU/SMP, Interrupts, Low-level I/O, Scheduler, Physical Memory Management)
  - ➔ *User-Mode Managers* (Tasks, Threads, Virtual Memory Management, Syscalls)

- ➔ *Communication Interfaces*

  The communication interface is one of the most important tasks of the kernel. At first the interprocess communication is a bottle-neck for the performance of the whole system. Secondly it's the only interface for the software to execute hardware and system management operations.

  - ➔ *Basic Interprocess communication* (asynchronous)
    - ➔ Every thread can register MessageHandler-Functions with named channels and act as receiver
    - ➔ Every thread can send data to named channels
    - ➔ Therefore every thread has for each open connection a MessageBus (Memory Region) through which the Messages can be passed to each other
    - ➔ There is no predefined protocol (with some exceptions)
  - ➔ *Exceptions* (a-/synchronous)
    - ➔ Every task can register an exception handler routine
    - ➔ A exceptions will cause the kernel to call the registered routines in user space
    - ➔ CPU Exceptions as well
  - ➔ Hardware communication (synchronous)
    - ➔ I/O-Port Access / DMA
    - ➔ IRQs
      - ➔ Special type of IPC (more likely RPC)
      - ➔ There can be only one InterruptRoutine per  Interrupt
  - ➔ Syscalls (synchronous)

- ➔ Platform

  This is a very basic hardware abstraction layer.

- ➔ *Debug Interface*

  *Maybe it might be useful to compile a basic Text-Mode driver with debug functions into the debug version of the kernel.*

**Goals**

- As small as possible
- As easy to understand as possible
- As easy to modify as possible
- As fast as possible
- As secure as possible

**Features**

- Multitasking
- SMP
- Paging
- Fast boot process

# I. Tasks

## a) Types

- Kernel (Special Task: The kernel itself)
- Drivers (Hardware management of the OS)
  - Only their threads can have real time priorities
  - Have hardware specific permissions
- Servers (Software management of the OS)
  - They handle everything virtual in the system like a VFS, HAL, Executables, Protocols, GUI
- Programs (User executables)
  - They have the lowest privilege level
  - This are the actual User Processes

## b) Task Class

Every type of task has a MainThread and possibly some ChildThreads.

A Task Class carries following data:

- Task ID
- Task Type
- Task State (Running, Waiting, Sleeping)
- Owner ID
- Virtual Address Space
- Exception Handlers
- Resources
- Shared Memory
- Private Kernel Stack

## c) Thread Class

Every thread has to carry this data:

- Global ID
- Local ID
- Priority
- Available CPU time
- Used CPU time

- Own Stack
- Own Message Buses
- Status Registers

## II. Resources

Managers provide Resources which can be requested by a thread.

The available Resources are:

- Memory (Request more memory of a specific type: Heaps, MessageBuses, etc.)
- Scheduler
- Threads
- Low-level I/O
- System Exceptions

## III. Datatypes

### a) Basic

| *Name* | *Type* |
|---|---|
| Byte | unsigned char |
| Word | unsigned short |
| Dword | unsigned long |

### b) Memory

| *Name* | *Type* |
|---|---|
| Address | Word |
| PhysicalAddress | Address |
| VirtualAddress | Address |

### c) Misc

| *Name* | *Length (Bit)* |
|---|---|
| size_t | Dword |

## Booting

The format of the kernel image will be a multiboot compatible Elf32-file.

**Bootloader**

As standard Bootloader this project will use GRUB 2. But every other multiboot compatible Bootloader should do it, too.

The task of the Bootloader is to load the kernel-image and, since the kernel has no device drivers, the Servers as modules.

## Kernel Class

### *I. Jobs*

- Bundle the Managers and initialize them
- Register own WorkerProcess (e.g. for asynchronous messages)
- Initialize the servers modules
- Starts the server processes

### *II. Definition*

| *Kernel* | |
|---|---|
| public | void Init() |
| | void Run() |
| | void GetVersion(Version *pVersion) |
| | void GetCodeName(char *pCodeName) |
| private | Version KernelVersion |
| | char CodeName[25] |

## Managers

Managers are the core modules of the micro-kernel, they and the Kernel Class, together, form the whole micro-kernel. This Managers will provide a standard interface to the kernel, so that it's simple to add/replace/remove individual Managers.

# *I. Memory*

## a) Physical Memory Management

The PhysicalMemoryManager Class depends on information provided by a BootInfo Class. That's because it needs a MemoryMap to initialize it's own. After this, the Manager knows where the free/usable memory is and is ready to use.

The algorithm will use two Bitmaps for Blocks (with the size of one Page) and SuperBlocks (with the size of SUPER_BLOCK_SIZE Pages). (Or maybe in combination with a stack)

### Structure

| *PhysicalMemoryManager* | |
|---|---|
| `public` | `PhysicalAddress AllocBlock(size_t Count = 1)` |
| | `void FreeBlock(PhysicalAddress Pointer)` |
| | `void ReserveArea(PhysicalAddress Base, size_t Size)` |
| | `size_t GetFreeMemory()` |
| | `size_t GetUsedMemory()` |
| | `size_t GetAvailibleMemory()` |
| `private` | `Bitmap Blocks` |
| | `Bitmap SuperBlocks` |

## b) Virtual Memory Management

The VirtualMemoryManager Class is needed to control the the VirtualAddressSpaces of the Kernel and Tasks. It will use paging with a page size of 4 kilobytes.

### Definitions

| *VirtualMemoryManager* | |
|---|---|
| `public` | `VirtualAddress AllocPage(size_t Count = 1)` |
| | `void FreePage(VirtualAddress Pointer)` |
| | `void IdentityMap(VirtualAddress Base, size_t Size)` |
| | `PhysicalAddress TranslateAddress(size_t DirectoryIndex, VirtualAddress Address)` |
| | `VirtualAddress TranslateAddress(size_t DirectoryIndex, PhysicalAddress Address)` |
| `private` | `List<PageDirectory> PageDirectories` |

## II. Platform

### a) Boot Information Management

The BootInformationManager Class parses important boot information passed by the bootloader and provides them to the Kernel.

**Definitions**

| *BootInformationManager* | |
|---|---|
| public | void Init() |
| | |
| | |
| | |
| | |
| private | size_t MemorySize |
| | char BootloaderName[25] |
| | char CommandLine[256] |
| | BootModules Modules[25] |

## *III. Multitasking*

### a) Scheduler Management

The SchedulingManager Class will use the round robin algorithm with priorities.

**Provides**

- InterruptRoutine (Checks if Thread/Context switch is needed)
- Thread & Context switch

### b) Task Management

The TaskManager Class stores and manages information about all running tasks and their threads.

**Provides**

- Get Task/Thread count
- Add Task/Thread
- Kill Task/Thread
- Pause Task/Thread

# *IV. Communication*

## a) Interrupt Management

The InterruptManager class is needed to receive interrupts and send a interrupt notification[...]

The notification system will be synchronous. The notification will be passed to the corresponding Scheduler-/Syscall-/ExceptionManager or User InterruptRoutine, first.

**Provides**

- Add interrupt routine
- IRQ Remapping

## b) Syscall Management

The SysCallManager Class is needed to process the Syscalls and execute the corresponding functions in kernel space.

**Provides**

- InterruptRoutine/FastSyscallRoutine
- The different Syscall functions

## c) Exception Management

The ExceptionManager is needed to process the CPU Exceptions and execute the corresponding functions in kernel space.

The Exceptions are handled synchronous:

Tasks can register a specific ExceptionHandler-Function which will be called when the exception occurs. There can only exist one ExceptionHandler-Function per Exception code.

**Provides**

- InterruptRoutine

## d) Interprocess Communication Management

The IpcManager Class controls the asynchronous part of the Communication layer.

Every Connection between a Client and a Server uses a MessageBus structure for the transfer, which is a shared memory area between the two communicating tasks. This area is split into two parts: Input (read-only) and Output (read/write).

The transfer of the messages is done by the tasks themselves: The Client and Server Task register a own Proxy Thread which manages the communication through a MassageBus.

The MessageBus struct is a template which could be used for every type of data, e.g.: Circular Buffers, simple structs, strings, binary data, etc.

This way the lowlevel IPC system is very flexible.

To establish a connection between two tasks a addressing system is needed. Though every task can register a channel with an arbitrary name. If a Client wants to connect to a Server it has to use the following addressing scheme: ServerName[:TaskID].ChannelName.

### e) Hardware Input/Output Management

The HardwareIoManager class manages the direct I/O-Port access and DMA.

**Provides**

- Port Input/Output functions
- DMA Mapping

## Synchronization Mechanisms

## Source directory tree

- Stage0: Very low level initialization. (e.g.: Assembler start code or Multiboot header/information)
- Stage1: C++ specific initialization code; Starts up the Kernel Class
- LibCpp: Some helper functions (e.g.: memcpy, strlen)
- Managers
    - Communication: Provides a communication layer to the system
    - Memory: Manages the system's virtual and physical memory.
    - Multitasking: Manages all the running tasks and threads.
    - Platform: Provides a very basic and low-level hardware abstraction layer.
    - Resources: Manages the system's Resources.
- Kernel
    - Kernel.[cpp/h]: Kernel Class

## The Servers

The servers aren't really a part of the kernel itself, they are more likely an interface between the user processes/drivers and the kernel.

i Add links to projects/sources by which I was inspired.
L4 API Specification [x.2, latest, May 15 2009]: http://hg.l4ka.org/l4ka-pistachio-ref/l4-x2.pdf
The X Operating System [May 15 2010]: http://free.prohosting.com/~xos/xos.html