Wrocław 22.02.2007

Project Report

# spirit

***Process Migration Layer and Abstract Node System for Distributed Systems Applications***

**Mateusz Styrczula** **Michał Kraszewski**

*"...the Linux philosophy is 'laugh in the face of danger'. Oops. Wrong one. 'Do it yourself'. That's it."*
Linus Torvalds, (1996-10-16).
Post to linux.dev.kernel newsgroup.

# The Project

***spirit*** *is a set of abstract tools (or, actually, an attempt to write them) enabling programmer to write a program that can migrate while running from one machine to another.*

# Table of Contents

# 1. Introduction

## About

**spirit**[1] is a programming project being in realization at Institute of Computer Science at Wrocław University as a part of Distributed Operating Systems lecture. Project's main goal is to learn how real operating system really works.

## Background

Developing an operating system is not a child’s play. It requires good knowledge of algorithms, data structures, programing techniques, optimization tips, low-level hardware designs and many more. Designing a new operating system is a very hard work requiring one to possess all of above combined with programming skills and intuition. Writing own fully-featured, supporting variety of hardware operating system from scratch is definetely overwhelming task. Nevertheless it is not necessary to write an operating system to understand how it is made. There are some possibilities helping curious man to achieve it. One of them is obvious – Linux! Unfortunatelly the Linux operating system kernel is complicated and complex programming project written by the programmers from all over the world. In fact, there is source code available, but finding out secrets of its functioning can pose some difficulty.

Writing an application that operates on processes requires understanding of general ideas on how they act in a multitasking environments, although is not sufficient – it is just a first step in grasping how the whole mechanism works. Despite of all above difficulties it is worth to make an effort to study this fascinating discipline because there are still many things to do. *“I will consider the job finished when no manufacturer anywhere makes a PC with a reset button. TVs don't have reset buttons. Stereos don't have reset buttons. Cars don't have reset buttons. They are full of software but don't need them. Computers need reset buttons because their software crashes a lot...”* (Andrew S. Tanenbaum).

## Process migration

Process migration is an issue being studied by scientists for many years. Yet we want to propose slightly less scientific approach. The

[1] The spirit is an immaterial but ubiquitous substance or energy present individually in all living things. Unlike the concept of human souls, which is believed to be eternal and preexisting, a spirit develops and grows as an integral aspect of the living being. **[1]**

vision of intelligent beings traversing enormous cyperspace is fascinating science-fiction writers for many years. Us too. So, let's imagine a computer program as a soul[2], that is, an immaterial creation, which may exist in many dimensions like in computer memory, on printed document or in human's thought... When thinking of its life-space, that is, in which it can operate and execute itself, the most natural association is operating memory of a computer system. The computer program itself does not make an autonomous unit; it is only an algorithm, a "recipe". It doesn't have its own state. The essence of a program are: its state, an algorithm, and an execution unit, therefore a running process can be considered as a living being[3]. It is well known that process consists of executable code and a state (variables, stack, heap and various flags). But even the smartest code is limited in its freedom. Its world is restricted by the boundaries of computer's operating system on which it is running[4].

## The purpose

The purpose of the **spirit** project is to free processes giving them the ability to spontaneously[5] pass from one machine to another[6]. To achieve this goal one should create fairly complex distributed system, built of many layers providing variety of services.

[2] The soul, according to many religious and philosophical traditions, is a self-aware ethereal substance particular to a unique living being. In these traditions the soul is thought to incorporate the inner essence of each living being. **[2]**
[3] The relationship between an execution unit (a processor) and coded algorithm is a loose analogy to the relationship between a spirit and a soul. There are many analogies to the world of living (and dead) in the UNIX-like systems. Processes are being killed, they can be in a zombie state, there are deamons etc. and it does not confuse anybody. Hence our parallel of process to living being.
[4] We think that writing this document in a fairy tale manner makes reading it more pleasant and allows us to explain complex things in a natural way. Wonderful examples of that style can be **[3]** and **[4]**.
[5] Spontaneously – Late Latin *spontaneus*, from Latin *sponte* of one's free will, voluntairly. **[5]**
[6] Reason for doing it is not necessary. The ability by itself is a reason good enough. We will discuss practical uses in chapter 3.

# 2. Project overview

```
+--------------------------------------------------+
|              "barking dogs system"               |
| or bindings to higher-level programming languages|  >- Example User-App
+--------------------------------------------------+
                         ^
                         |
                         v
+---------------------"Spirit"---------------------+          --+
|                                                  |            |
|  +-------------Abstract Node System-----------+  | -+         |
|  |+------------------------------------------+|  |  |         |
|  ||           upper layer libraries          ||  |  |         |
|  |+------------------------------------------+|  |  |         |
|  |+---Node---+  +---Node----+     +---Node---+|  |  +- UPPER  |
|  ||  alpha 1 |  |  alpha 2  | ... |  alpha N ||  |  |  LAYER  |
|  |+----------+  +-----------+     +----------+|  |  |         +- MIDDLEWARE
|  +--------------------------------------------+  | -+         |
|                        ...                       |            |
|  +-----------Process Migration Layer----------+  | -+         |
|  | (modules, daemons, lower layer libraries)  |  |  +- LOWER  |
|  +--------------------------------------------+  | -+  LAYER  |
|                                                  |            |
+--------------------------------------------------+          --+
         ^              ^                 ^
         v              v                 v
+------------Connected Systems (TCP/IP)------------+
|  +-----------+  +-----------+     +-----------+  |
|  | 1st Linux |  | 2nd Linux | ... | Nth Linux |  |  >- Network OSes
|  +-----------+  +-----------+     +-----------+  |
+--------------------------------------------------+
```

We will discuss lower layer of **spirit** system (Process Migration Layer). It consists of:

- kernel module (Linux version 2.6.x and above),
- standard C library routines for Process Migration Layer,
- user-space daemon responsible for transferring processes,

## Goals

We intended to write system allowing process transfer between different machines allowing it to maintain state. System is divided into three major parts described below.

- **kernel module:**
  - interface allowing communication between user-space programs and kernel functions ***(achieved)***,
  - functions:
    - processor registers dump ***(achieved)***,
    - structures describing process' state dump[7] ***(achieved)***,
    - structures describing process' memory dump ***(achieved)***,

[7] Process state in this context means Linux `task_struct` and `mm_struct` structures.

        - opened files dump ***(to do)***,
        - established network connections dump ***(to do)***,
        - signal queues and handlers dump ***(to do)***,
        - terminal state dump ***(to do)***,
        - shared libraries (state and libraries itself) dump ***(to do)***,
    - binary file format and handler ***(achieved)***,

- **standard C library:**
    - kernel module communication interface ***(prototype)***,
    - transfer daemon communication interface ***(prototype)***,

- **user-space daemon:**
    - process transfer between machines ***(achieved)***,
    - required open files transfer ***(achieved)***,
    - missing shared libraries transfer ***(to do)***,
    - restoring process environment on target machine ***(to do)***.

# 3. Possible applications

Apart from the whole debate about souls and spirits (considered from computer industry point of view is, to put it mildly, at least useless), one can imagine some scenarios of practical usage of elements of process migration system. Some of examples could be:

- Internet, connections between computers, local area networks and metropolitan area networks are systematically increasing bandwidth. Distributed internet applications are very popular these days. Everybody has an oportunity to download an operating system or anti-virus program update. Local networks are almost everywhere. It is not so hard to imagine an general-mainteneance program as an agent traveling through local network and performing some administrative tasks (eg. anti-virus scanning, disk-space optimization, updating necessary software etc.)
- Closed-source program licenses. Image processing, office or professional engineering software are usually expensive. Let’s imagine a user having a computer at home and in his office. If a large scale process migration would be possible, such user could have his favorite programs set in a one copy (with one license)

and use them in different places (launching them on one computer and calling them to place where he is currently working).

- Multi-core processors are common. It is very likely that single chips of large scale integration will be manufactured with increasing number of intependent processing units. Some aspects of process migration between machines will have reflection in "micro-world" of multi-core processors. It would be more legitimate thinking about it in the context of operating system build upon some micro-kernel architecture. In such architecture data exchange is based on message passing instead of common shared memory. It is proven to be more secure and stable way to achieve highly reliable, self-healing operating systems[8]. Breakdown of one component do not have influence on general system stability. What in monolithic-kernel based system is a "critical" process, in a micro-kernel one not necessairly is. Message passing between units of multiprocessor system, when no shared memory access is possible between different processing units, does not fundamentally differ from message passing between processes, even in a very wide area networks. Process migration knowledge could be used to reduce the amount of data passed between different execution units by transferring processes that „talk to each other intensively" to one execution unit allowing them to „discuss" via common shared memory.

[8] It is Andrew S. Tanenbaum point of view. **[6]**

# 4. Issues[9] (future work)

Many hours of pleasant work are behind us. Basic functionality is achieved, but there is still much to be done. This includes following things:

- full and stable working dump of files, message queues, pipes, sockets, process current working directory, signal queues, handlers and terminal state,
- dump-file version checking on transfer,
- shared library support and apropriate `load_shlib` callback in `linux_binfmt` structure,
- shared library versioning and transferring on demand,
- stable and well-documented unified standard C libraries providing needed interfaces and functions to manage process transferring tasks,
- searching for needed resources on target host first and then transferring them if necessary,
- support for user permissions other than root,
- environment rebuilding before process restart,
- build in security mechanisms (process access control, user permissions),
- sandbox or some kind of playground for incoming processes,
- host system load control and process transfer permissions management,
- move as many components to user-space as possible,
- backing up processes and restoring from backup after process execution failure on target host,
- build upper layer (Abstract Node System), description of which is far beyond this paper’s subject,

[9] Yes. It may seem, that looking at what is still to be done there is hardly anything done already. Although there are many things to be done in the subject solid foundation has been made and most of described issues has been investigated.

# 5. Overview of project source code

```
.
|-- common
|   |-- defines.h           - common project definitions,
|   |-- ioctl_commands.h    - IOCTL commands definitions,
|   |-- messages.h          - communication protocol between kernel module,
|   |                         and user-space programs,
|   |-- structures.h        - structures describing spirit binary file format,
|   |                         tasks, memory and cpu-registers,
|   `-- version.h           - project authors, name, version, license type etc.,
|-- d_mon
|   |-- trans
|   |-- Makefile
|   |-- dlib.c              - library providing simple interface allowing
|   |                         communication with transfer daemon,
|   |-- dlib.h
|   |-- dmon.c              - file containing main() function responsible for
|   |                         initialization of transfer daemon and pasing
|   |                         control to connection thread,
|   |-- dmon.h
|   |-- dmon_chthrd.c       - connection handling thread - controls particular
|   |                         connection with daemon,
|   |-- dmon_chthrd.h
|   |-- dmon_ctl.c          - functions controling daemon bahavior, like
|   |                         signal actions, and definitions of control
|   |                         structures,
|   |-- dmon_ctl.h
|   |-- dmon_ioctl.c        - stub for kernel module control library,
|   |-- dmon_ioctl.h
|   |-- dmon_main.c         - functions responsible for handling incoming
|   |                         connections and pasing control to proper
|   |                         connection handling thread,
|   |-- dmon_main.h
|   |-- dmon_msg.c          - unified protocol message functions, allowing to
|   |                         send and recieve messages, used for both client
|   |                         and server side communication,
|   |-- dmon_msg.h
|   |-- protocol.h          - communication protocol definition,
|   |-- spirit01.c          - prototype of traveler,
|   `-- zclient.c           - prototype debug interface, allows sending control
|                             commands to daemon,
|-- ioctl_tester
|   `-- ioctl_tester.c      - a "Swiss Army knife"[10] and remote control to every
|                             single one functionality which is provided by
|                             spirit kernel module. It's capable of perform whole
|                             dump, restart process and also examine the contents
|                             of dump files and kernel module "health".
|-- kertinker
|   |-- Makefile
|   |-- actions.c           - spirit kernel module work-horse
|   |                         function definitions,
|   |-- actions_externs.h
|   |-- initexit.c          - init and exit module function definitions,
|   |-- k_mtg.c             - first draft - testing new ideas, notes, misc...
|   |-- kertink.c           - IOCTL handling function set,
|   |-- kertink_externs.h
|   |-- misc_from_kernel.c  - miscellaneous functions from Linux kernel
|   |                         that are, unfortanetely, not exported,
|   |-- module_state.h      - module operations state structure,
|   |-- variables.c         - global module variable definitions,
|   `-- variables_externs.h
`-- LICENSE                 - GPL license text,
```

[10] A Swiss Army knife (SAK) is a multi-function pocket knife or multitool. The term "Swiss Army knife" is sometimes used generically to describe a tool, such as a software tool, that is a collection of special-purpose tools. **[7]**

# 6. Kernel module internals

**kertinker/init_exit.c**

- `spirit_module_init ()` - program's "entry point", registers device `/dev/kertinker` in system, registers spirit file binary format, allocate memory for internal buffer, message-passing interface and module-state structure,
- `spirit_module_exit ()` - executed when module unloading is performed. Deregistering device, binary file format, freeing allocated memories

**kertinker/kertink.c**

- `kertink_device_open ()`, `kertink_device_release ()` - responsible for handling device's opening and closing events,
- `kertink_device_ioctl ()` - called whenever a process tries to do an IOCTL, responsible for calling apropriate "action" handler,

**kertinker/actions.c**

- `kertink_mp_handle ()` - called from within `kertink_device_ioctl ()` described above. It is bi-directional (kernel-space to/from user-space) message passing handling function,
- `mp_dump_processor_registers ()` - responsible for dumping process processor general purpose registers,
- `mp_dump_task_struct ()`, `mp_dump_mm_struct ()`, `mp_dump_thread_struct ()` - responsible for dumping (most significant parts of) task_struct, mm_struct and thread_struct structures,
- `mp_dump_vm_areas_reset ()` - reset vm_areas dumping sequence,
- `mp_dump_vm_bounds ()` - dump one vm_area_struct structure,
- `mp_dump_vm_data ()` - dump one "chunk" of process memory,
- `mp_dump_vm_filename ()` - if memory actually being dumped is a file mmaped[11] in memory, obtain that file name,
- `spirit_restart ()` - responsible for restart process from file - registered as handler to spirit binary format, invoked by kernel itself when any process in system tries to execute a file. This function is in fact the core of kernel-side tasks of spirit project.

[11] **[8]** pages 116 to 117 and **[9]** pages 290 to 293

**kertinker/misc_from_kernel.c**

- `access_process_vm ()` - allow access to another process' address space (read/write),

**common/defines.h**

- `SPIRIT_MAGIC_N` - definition of „magic numbers” used to recognize **spirit** binary format,
- `DEVICE_MAJOR_NUM` - used device major number,

**common/messages.h**

- `struct messpass {...}` - message passing structure definition,

**common/structures.h**

- `struct spirit_binfile_header {...}` - **spirit** binary format file header description,
- `typedef enum {...} spirit_data_type_t` - binary file consists of blocks. Every block is of type described in this enumeration.
- `struct spirit_block_header {...}` - every data block is preceded by this header.

# 7. User-space daemon internals

**d_mon/dmon.c**

- `main()` - initialize control structures, assign signal handling functions, daemonize and start a thread awaiting incoming connections, after that begin signal handling loop,
- `on SIGTERM` - break the signal loop, signal the main connection thread to stop and wait while it stops any active connection handling threads, free control structures and quit,

**d_mon/dmon_main.c**

- `dmon_main_proc()` - create and bind a server socket to listen for incoming connections, enter loop passively waiting for connections and spawning new threads to handle them,

**d_mon/dmon_chthrd.c**

- `connection_thread()` - await for messages, and process requests from client,

*Programs are written in GNU dialect of C programming language (which is standard of developing Linux kernel and is an extension to ANSI standard). We made an effort to make the source code fulfill requirements of „well formatted and legible knowledge piece”. It is written to the best of our abilities to be self-explanatory and richly commented. We encourage to read definitions of the above functions and trace the program control flow.*

# 8. Literature

Following books and materials helped us a lot in every stage of project developing during near half a year of work. Some of them are strictly technical and some are general operating systems knowledge. They are solid knowledge foundation.

**[1]** ***Wikipedia The Free Encyclopedia (accessed 22.02.2007)***
*http://en.wikipedia.org/wiki/Spirit*

**[2]** ***Wikipedia The Free Encyclopedia (accessed 22.02.2007)***
*http://en.wikipedia.org/wiki/Soul*

**[3]** ***Randal L. Schwartz, Tom Christiansen: Perl Wprowadzenie.***
*Wyd. 2. Gliwice: HELION, 2000. ISBN 83-7197-220-2*
Słowo wstępne, s. 9-11

**[4]** ***Eric S. Raymond: UNIX Sztuka programowania.***
*Gliwice: HELION, 2004. ISBN 83-7361-419-2*

**[5]** ***Merriam-Webster Online Dictionary (accessed 22.02.2007)***
*http://www.m-w.com/dictionary/spontaneous*

**[6]** ***Tanenbaum-Torvalds Debate: Part II (accessed 22.02.2007)***
*http://www.cs.vu.nl/~ast/reliable-os/*

**[7]** ***Wikipedia The Free Encyclopedia (accessed 22.02.2007)***
*http://en.wikipedia.org/wiki/Swiss_Army_knife*

**[8]** ***Neil Matthew, Richard Stones: Linux Programowanie***
*Warszawa: RM, 1999. ISBN 83-7243-020-9*
Rozdział 3: Praca z plikami, s. 79-119
Rozdział 4: Środowisko uniksa, s. 121-155
Rozdział 5:Terminale, s. 157-190
Rozdział 7: Zarządzanie danymi, s. 233-258
Rozdział 8: Narzędzia programistyczne, s. 281-301
Rozdział 10: Procesy i sygnały, s. 347-378
Rozdział 12: Semafory, pamięć dzielona i kolejki komunikatów, s. 431-447
Rozdział 13: Gniazda, s. 461-494

**[9]** ***Robert Love: Linux Kernel Przewodnik programisty.***
*Gliwice: HELION, 2004. ISBN 83-7361-439-7*

***Linus Torvalds: Linux Kernel Source Code version 2.6.17***
*http://www.kernel.org/*
*http://www.gelato.unsw.edu.au/lxr/source/?a=i386*

***Mark Mitchell, Jeffrey Oldham, Alex Samuel:***
***LINUX Programowanie dla zaawansowanych.***
*Warszawa: RM, 2002. ISBN 83-7243-217-1*

***Neil Matthew, Richard Stones:***
***Zaawansowane programowanie w systemie Linux.***
*Gliwice: HELION, 2002. ISBN 83-7197-495-7*
Rozdział 26: Sterowniki urządzeń, s. 1019-1055

***Moshe Bar: LINUX systemy plików.***
*Warszawa: RM, 2002. ISBN 83-7243-256-2*
Rozdział 3: Co to jest system plików?, s. 19-66
Rozdział 4: Wirtualny system plików Linuksa, s. 69-117

***B. W. Kernighan, D. M. Ritche: ANSI C.***
*Wyd. 7. Warszawa: WNT, 2002. ISBN 83-204-2719-3*

***A. Silberschatz, P. B. Galvin, G. Gagne:***
***Podstawy systemów operacyjnych.***
*Wyd. 6. Warszawa: WNT, 2005. ISBN 83-204-2961-7*

***A. S. Tanenbaum, M. van Steen:***
***Systemy rozproszone Zasady i paradygmaty.***
*Warszawa: WNT, 2006. ISBN 83-204-3070-4*

***D. P. Bovet, M. Cesati: Understanding the Linux Kernel.***
*O'Reilly, 2000. ISBN 0-596-00002-2*

***Michael Beck i in.: Linux Kernel Internals,***
*Wyd. 2. Addison Wesley, 1998. ISBN 0-201-33143-8*

***Mel Gorman: Understanding the Linux Virtual Memory Manager.***
*http://www.csn.ul.ie/~mel/projects/vm/guide/pdf/understand.pdf*

***Mel Gorman:***
***Code comentary on the Linux Virtual Memory Manager***
*http://www.csn.ul.ie/~mel/projects/vm/guide/pdf/code.pdf*

***Paul Rusty Russel: Unreliable Guide To Locking.***
*http://people.netfilter.org/~rusty/unreliable-guides/kernel-locking/kernel-locking.docbook/lklockingguide.html*

***Peter Jay Salzman, Michael Burian, Ori Pomerantz:***
***The Linux Kernel Module Programming Guide.***
*http://www.dirac.org/linux/writing/lkmpg/2.6/lkmpg-2.6.0.html*

***Alessandro Rubini: Kernel System Calls.***
*http://www.linux.it/~rubini/docs/ksys/*