Phantom OS Documentation

Release 0.1-0

Dmitry Zavalishin

Jan 16, 2020

Contents

CHAPTER 1

Introduction

This book is Phantom OS developer's guide. It contains description of OS internals and examples on writing kernel components and userland code.

Source codes for OS can be found in Phantom OS GitHub repository.

Source for this book itself are in book GitHub repository.

1.1 What Phantom OS is

To be short:

- Orthogonal persistence. Application does not feel OS shutdown and restart. Even abrupt restart. It is guaranteed that application will be restarted in consistent state, which is not too old.
- As long as you have reference to any variable, it's state is the same between OS reboots. You don't have (though you can) save program state to files. It is persistent.
- Managed code. Native Phantom applications are running in a bytecode machine. (But it is worth to mention that Phantom has simple Posix compatibility subsystem too.)
- Global address space. Phantom OS is an application server. All applications can communicate directly, by sharing objects.

Phantom OS persistence is achieved not by serializing data to files, but by running all applications in a persistent RAM. You can (and it will be true) think of Phantom memory subsystem as of a persistent paging engine. All the memory is paged to disk in a way that lets OS to restore whole memory image on restart. Consistently.

Which subsystems Phantom currently includes:

- Kernel itself: threads, synchronization, persistent memory management.
- Bytecode virtual machine - running native applications.
- Posix layer - runs Linux compatible (but not yet persistent) code.
- Graphics subsystem - Windows, controls, UI.

- Networking (TCP/IP)
- Phantom language compiler - the most native userland language
- Java to Phantom translator - work in progress
- Python to Phantom translator - just started

Technical information and links:

- Source code
- Wiki
- Web site

1.1.1 Do we need one more OS

There are millions of homegrown operating systems around. Why one more?

Well, here I will try to describe the reasons behind making Phantom OS.

Operating system of these days is, actually, all that is seen by the appication. It is, in fact, a virtual computer an application is running on. As such, operating system can create nearly any kind of environment for the program.

But due just to historical reasons traditional operating systems keep to be very thin wrap around the CPU providing just some drivers and libraries.

One of things that is quite possible but never even thought about is ability to hide from the program fact of stop and start of OS kernel. It is not very hard to achieve, and changes situation dramatically.

Having this as a goal one can design a persistent environment, which changes game rules quite a lot. But! Program running in a persistent OS can completely ignore the difference between persistent and usual OS environment, not forcing programmer to learn new tricks.

What is good about persistent environment?

You don't need files any more

Really. The file is just a tool which helps program to survive for the time computer is being turned off. If operating system hides this situation from you, there is no need to save anything to file. Any variable is... like file now? It just keeps its value forever.

What's more interesting - all your complex data structures don't have to be serialized. You are not limited to structure which simplifies serializing, and don't have to write corresponding code.

Well, one can say that you still need to be able to save data to file to interchange data with traditional software. Yes and no.

- You don't have to do it from the very beginning - write your program in a simple way and add file operations later.
- You don't have file interface to be complete - sometimes partial save to file is ok. For example, program settings and configuration need not to be saved in files at all.
- Nowadays a lot of interchange is being done via the Internet API, which is different kind of fish.

Don't recreate the environment

Think of the program which works with a lot of, say, TrueType fonts. In classic OS it takes a lot of time each time it starts to build list of fonts available scanning files and parsing data structures again and again.

In persistent OS list of fonts you built once is available for you forever. Just in a form that is ideal for your program.

Don't recreate results

In traditional OS every program with UI is a complex combination of code that builds program state and code that has to re-visualise it through the UI.

Lets imagine we have to paint day to day temperature curve in traditional OS.

We will need:

- Code that gets new measurements and stores data to database
- Code that extracts data from database
- Code that paints dataset in an OS window
- Control logic to orchestrate all these parts together
- Some deployment rules and glue settings for program to find its database

Now for the persistent OS:

- Code to read new measurement and put one pixel to the window.

And... that is all. Really.

Window is persistent too, you never have to repaint it. As a result - you do not need to store data at all! No repaint - no storage. No database, no deployment rules, no setup. Just 5-10 lines of code.

What Phantom OS is going to be - an environment where simple goals can be achieved in a simple way.

Phantom OS lifecycle

Persistent OS lifecycle is differs a bit from traditional OS.

We differentiate between initial OS instance start and all of the subsequent kernel starts, which we call *restarts*.

There is also an events in the usual life of the kernel called *snapshots*. These events store consistent image of OS userland programs on disk. If kernel will be stopped abruptly, the next *restart* will continue executing user programs from last successful *snapshot* point.

As long as *snapshot* is being done and is not finished completely OS keeps previous two snapshots intact. After the new snapshot is made the most old one is released.

It means that at any point in time there is at least one complete snapshot available on disk. Most of the time there are two.

When kernel restarts it is possible to choose if we restart from latest or previous snapshot.

Technically it is possible to keep more snapshots, like, for example, weekly or monthly ones.

In fact, snapshots engine at the same time is incremental backup engine.

When Phantom OS instance boots for the first time in its life and has no snapshot to continue from, it has to create basic initial state. It includes initial set of classes in object land and some initial code running. This code is supposed to set up user environment and bring applications to OS instance.

It is similar in some ways to Unix `init` process, but `init` has to set up OS environment at every start of kernel, which is not needed in Phantom.

Note that all of that is true for native Phantom OS *personality*. There is also POSIX *personality*, which currently is not persistent.

1.1.2 Phantom OS Architecture

Basic kernel engines

Basic engines provide lowest level services for functional kernel subsystems and create hardware independent abstractions.

Threads Preemprive multitasking for kernel threads.

Scheduler Thread priorities, including real-time.

Mutex/Cond/Sem And spinlock as lowest level primitive. Provide threads syncronization and event signalling.

Physical memory allocator RAM allocation. Co-operates with paging engine to keep supply of free memory by paging memory out. Also includes address space allocator which can be used to create memory windows or memory page aliases.

Bus IO Also PCI enumeration. Provides for low level hardware communications.

Interrupts and exceptions Hardware interrupts, context switching, page faults.

Page table Main low-level engine of persistent memory.

Cbuf,KHash Basic kernel data structures.

Pools Reference counting kernel object containers.

Timing Time of day, timer callouts, etc.

Main kernel subsystems

These subsystems are either visible to userland code through some API (like, for example, TCP/IP), or provide seamless services - persistent memory is obvious example.

Drivers Of course, this is the lowest level which can be directly visible to user code. Drivers are divided into block (disk), character, network and graphics drivers. Sound device driver is, for example, a character driver.

Please note driver/device properties engine. It's a phantom specific ioctl like interface.

Network stack One of the basic building blocks of modern kernel. Note that Phantom is able to set up paging over the network.

Graphics/Windows/Controls/Events There is an integrated kernel-level windowing subsystem in Phantom OS. TrueType fonts, bitmap fonts, set of UI components, UTF-8 and -32, etc.

Main disk IO stack Asyncronous disk IO based on request objects. Partitions, sync IO interface, deep integration with paging system, cache support engine, all the stuff.

Unix emulation layer It is possible to compile Phantom kernel with Unix support turned off, but if you want files - Unix part is a point for filesystems to live in. Is able to run more or less POSIX-compatible Elf executables.

Persistent pager And snapshots subsystem - this is the heart of Phantom OS. Makes native application code to believe that OS never reboots.

Internal classes implementation A channel from object land to kernel API.

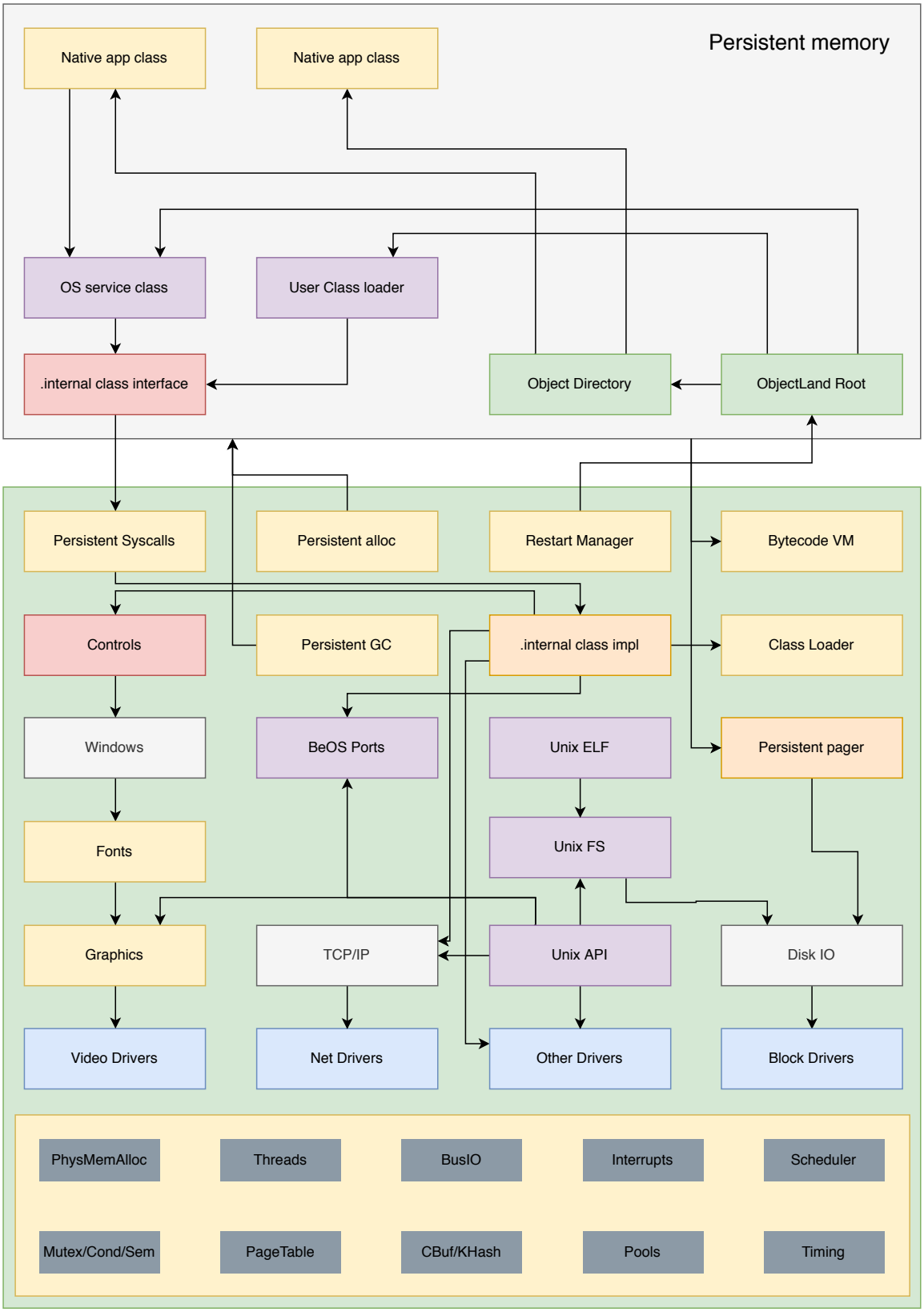

Fig. 1: Major components of Phantom OS instance
From bottom to top are basic kernel engines, main kernel subsystems and userland objects.

Persistent syscalls An engine that resolves problem of kernel reboot during a long syscall from persistent program.

Restart manager Provides means to find all objects that have to be re-connected to kernel on reboot.

Bytecode VM Virtual machine interpretator - executes userland bytecode.

Persistent memory allocator and GC Even persistent memory should be allocated.

Kernel class loader Used to initiate object land on the first boot of empty OS instance.

Native Toolchain

Phantom OS programs can be written in its own language which is directly supported by `plc` (phantom language compiler) or in Java or Python (this part is experimental yet).

For Java and Python compilation is done in two stages. First native parser compiles code and produces some intermediate representation, then `plc` consumes it and generates Phantom bytecode.

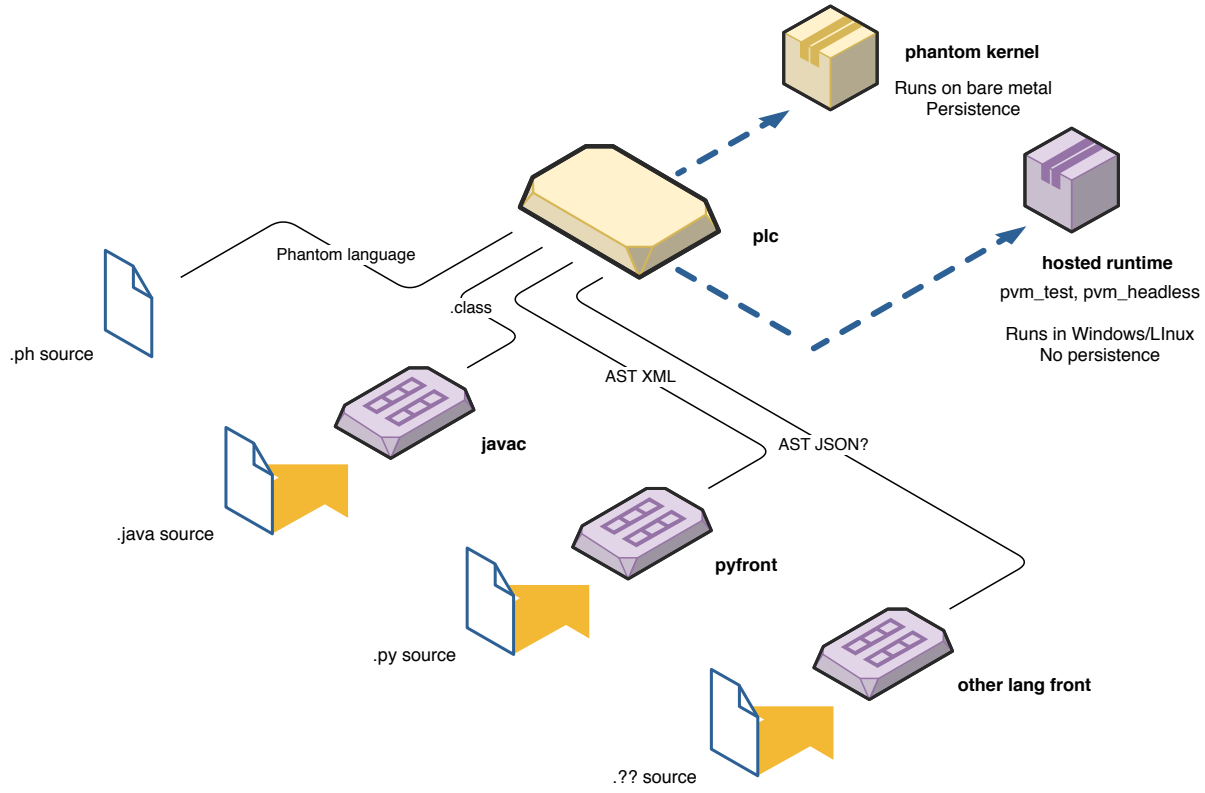

Fig. 2: Native toolchain of Phantom OS
Phantom bytecode can be run by Phantom OS kernel or by hosted runtime. In the last case persistence is not (yet?) supported.

Example of Phantom program

Lets review a simple Phantom OS program that gets weather information from public service and draws a diagram of temperature with the one minute step.

```
package .ru.dz.demo;

import .phantom.os;
import .internal.window;
import .internal.bitmap;
import .internal.tcp;
import .internal.time;
import .internal.directory;
import .internal.long;


class weather
{
    var xpos : int;
    var ypos : int;

    var win   : .internal.window;
    var sleep : .internal.time;
    var http  : .internal.tcp;

    var itemp : .internal.long;


    void run( void )
    {
        var bmp         : .internal.bitmap;
        var bmpw        : .internal.bitmap;
        var json_string : .internal.string;
        var json        : .internal.directory;
        var jtmp        : .internal.directory;

        bmp = new .internal.bitmap();
        bmp.loadFromString(getBackgroundImage());

        win = new .internal.window();

        win.setWinPosition(650,500);

        win.setTitle("Weather");

        win.setFg(0xFF93CDB4); // light green
        win.setBg(0xFFccccb4); // light milk

        win.clear();
        win.drawImage( 0, 0, bmp );

        bmpw = new .internal.bitmap();
        bmpw.loadFromString(import
            "../resources/backgrounds/weather_sun_sm.ppm");
        win.drawImage( 17, 102, bmpw );
        win.update();

        sleep = new .internal.time();
        http = new .internal.tcp();

        xpos = 17;
        while(1)
```

(continues on next page)

(continued from previous page)

```
        {
            json_string = http.curl(
                "http://api.weather.yandex.ru/v1/forecast?extra=true&limit=1",
                "X-Yandex-API-Key: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx\r\n" );

            json = json_string.parseJson();
            jtmp = json.get("fact");
            itemp = jtmp.get("temp");

            win.drawImagePart( 0, 0, bmp, 250, 240, 120, 22 );

            win.setFg(0); // black
            win.drawString( 280, 235, itemp.toString() );
            win.drawString( 250, 235, "T =" );

            ypos = 15 + (itemp * 2);

            win.setFg(0xFF93CDB4); // light green
            if( ypos < 70 )
                win.fillBox( xpos, ypos, 2, 2 );


            sleep.sleepSec( 60 );

            xpos = xpos+1;
            if( xpos > 358 ) xpos = 17;
        }
    }

    .internal.string getBackgroundImage()
    {
        return import "../resources/backgrounds/weather_window.ppm" ;
    }

};
```

`import .internal.window;`

Import statements bring in system service classes, such as window.

`win = new .internal.window();` Create instance of window. That is a canvas for us to paint on. In Phantom there is no paint() or repaint() request or message that forces us to repaint window contents. Persistent paradigm dictates direct way of communication: program paints into the window whenever it is needed and window contents are saved even across the OS reboots.

`win.setBg(0xFFccccb4);` Do some settings for window, such as foreground and backgroud colors, clear it, paint background image.

Now, here is quite an intersting part:

```
bmpw = new .internal.bitmap();
bmpw.loadFromString(import
    "../resources/backgrounds/weather_sun_sm.ppm");
win.drawImage( 17, 102, bmpw );
```

We create a variable of `.internal.bitmap` type, which is suitable for holding a bitmap image.

An `import ".../rweather_sun_sm.ppm"` construct is specific for Phantom language and creates a string constant which contains whole contents of host computer file.

Phantom OS is a system which can work completely without file system. If program needs some resource for its work, the resource can be just stored in a variable. An `import` operator creates such a variable with an asset (file contents) inside.

But it contains a bitmap in a `.ppm` file format, and we need a bitmap object which can be paint onto the window.

Here goes `bmpw.loadFromString()`, which can parse image file fomat from string and convert it into the internal representation.

This internal representation will continue to exist as content of `bmpw` variable.

Note that:

- We keep assets in program variables, not in files. It is safer as there is absolutely no access to those assets from outside of program. No virus or malware attack is possible as there is no way to reach our `bmpw` variable from outside.

  It is not so critical for a bitmap, but suppose it is not a bitmap but a Lua script? Or just a program code? Note that executable code of program is not a file too, it is a persistent object that is visible just to Phantom executive subsystem. You can not search filesystem and modify `.exe` file to implant virus code.

- We brought asset in as an image of some file, which is kept in a string constant. But this constant won't consume any memory or disk space as soon as program started, because it will be processed by `bmpw.loadFromString()` and freed by garbage collector.

- What we will keep as persistent variable value is internal representation of bitmap, which is instanly ready for use. After reboot or any number of reboots it will not be needed to re-read image file contents, parse file and convert it to bitmap.

  Any asset that is needed by Phantom program can be kept in a persistent variable just in a form that is needed by program.

- You don't have to provide a way to build distributive and install a program, deciding about which directory it should occupy on target machine and how it will find its assets. All the assets are inside. For this program just **one** class image is all the distributive. Well, for a more complex program it will be a set of classes (combined in a container, of course), but still there will be no problems with locating assets on a target file system.

Note: Strings are binaries.

Phantom .internal.string class can contain any binary content including binary zeroes and sometimes is used as a container for complex data or just as a byte buffer.

Ok, lets go furter.

```
win.drawImage( 17, 102, bmpw );
```

We paint our bitmap held in a persistent variable into the persistent window.

Note that this operation is performed just one time in this program's life too. The program will start and work as long as user needs it - practically until it will be killed by hand.

Image paint by this line of code will exist in a tagret window forever or as long as it is not overpaint by other program statements.

```
sleep = new .internal.time();
```

Here we create an instance of class that provides us with OS timing API. Later it will be used to sleep for a minute: `sleep.sleepSec( 60 );`.

```
http = new .internal.tcp();
```

This object is a handle to network services. Despite the name this class provides not TCP only, but some higher level API.

Note: Forever means forever in Phantom

As it is usually said, `while(1)` is a `forever()` statement.

In Phantom OS it is **literally** true. This loop will **never** end for the rest of this program's life. Again, as long as user does not kill the instance of this program.

```
json_string = http.curl( url, header )
```

Here we ask OS default HTTP client to retrieve some web content. As you can guess, in this case server returns us a JSON structure with current weather infiormation.

```
json = json_string.parseJson();
```

Parsing a JSON is a method of string class, because nowadays it is really frequently used thing. The returned object is of `directory` class, which is simply a hash map inside.

```
jtmp = json.get("fact");
```

Extract JSON child node. It is a hash map too.

```
itemp = jtmp.get("temp");
```

Extract numeric value of node.

```
win.drawImagePart( 0, 0, bmp, 250, 240, 120, 22 );
```

Paint part of background image to clear old temperature value string. Note that background image is kept in persistent variable too.

```
win.drawString( 280, 235, itemp.toString() );
```

Paint new temperature value as sting.

```
win.fillBox( xpos, ypos, 2, 2 );
```

Paint next point on temperature diagram.

Note again that we don't have to keep historical data for temperature and repaint diagram on program restart: we are in a persistent OS and previously paint data just keep exist in a window.

Of course, this program is (intentionally) extremely simpe, it, for example, does not scroll diagram image (though that is another 2 lines of code) and does not check for exceptions, so it will die on any incorrect JSON received. But that is another 2 lines of code too.

One could ask, is it ok not to keep historical temperature data at all, and it is a good question.

First of all, it is not a complex thing in Phantom at all. Just add `tempHistory = new .internal.array()` and put data in with `tempHistory.append( itemp )` - and you will get, practically, a database. Yes, one variable in Phantom OS can be a database. Simple one, sure.

But, frankly, we just don't need it for *this* program. It's goal is to paint diagram, and it does. There's no reason for a program to be more complex, that it has to.

That is a basic idea of Phantom OS programming paradigm. Simple things must be implemented in a simple way.

1.1.3 Frequently Asked Questions

Q: Traditional program starts and stops and when stopped, code of program can be updated, a new version of program intalled. If program does not stop, how can you update program code?

A: This question is discussed in a separate chaprer, please refer to *Program update*.

Q: In classic OS we can restart program if it fails. In persistent OS program will run forever and if it failed, it can not be restarted?

A: Program can be restarted in persistent OS just as you wish. The difference is that operating system does not kill program on each reboot. It does not forbid user to kill and re-run any program.

Q: Other systems will use files for long time. How can Phantom communicate with them if Phantom does not use files?

A: Phantom lets you write native programs that can work without files. But still you can use files. Actually, application program can ignore all the Phantom OS special features and work as if it was in traditional OS. Or use Phantom specific features partially, or mix traditional and Phantom OS style.

There are special services in Phantom which simplify writing programs that need to interoperate with others.

Q: There soon will be hardware RAM persistence in all computers and Phantom will not be needed, every OS will become a persistent OS.

A: It is not that simple. There are problems that must be solved by any OS to really hide a reboot from software. This book will give some ideas about these problems and solutions impemented in Phantom.

Q: Phantom persistent memory is a paging subsystem. It is processing disk IO in a way that is not controlled by application. Will it lead to performance degradation?

A: It is usual for modern software to suppose that work set of program is much less than amount of available RAM. It is not supposed that program will heavily rely on paging system to move pages to and from disk. In this situation Phantom's interference with program will be negligible.

One more thing to note is that amount of disk IO in persistent OS depends on amount of modified memory. For hi-performance software such as games memory modified is a game state, which is not really huge and is modified more or less in place. Good design of persistent memory allocator can insure high affinity of modified memory regions. It means that snapshot footprint won't grow even if snapshots will take longer than usual.

CHAPTER 2

Project structure

2.1 Directory tree

GitHub project directory tree:

oldtree Kernel source code

phantom Kernel libraries and subsystems source code

dev

Device drivers. Drivers moved here are more or less stable. Drivers that are still in development are in kernel src dir.

libc

This libc is used for kernel code.

libkern

Kernel infrastructure components that can't be used in user mode.

libphantom

Kernel components that are used in usermode virtual machine builds.

libwin

Graphics, windows, controls.

modules

Loadable kernel modules. Mostly experimental.

threads

Kernel threads subsystem. Also sync code - mutex/cond/sema.

user

Simple usermode support for POSIX subsystem.

vm

Last not least: bytecode virtual machine, code of native Phantom personality.

plib Phantom user mode libraries

run Setup for running Phantom OS in QEMU

test Regress tests for various subsystems

plc

Regress test for *plc* compiler. There are source files and corresponding compiler listings produced with last stable version of compiler.

New compiler is used to produce its listings and lstings are compared to reference ones.

You can check differences and decide if new verion is correct. There are two listings - *lst* and *lstc*. Former one dumps structure of internal compiler's AST like tree with node data types and relations. Latter one is an output bytecode listing.

unit

Unit tests for kernel C libraries.

external

These tests are supposed to connect to running kernel by TCP/IP. Implemented as JUnit tests.

tools Non-kernel code, tools to support OS programming. Notably here is *plc*, Phantom Language Compiler.

plc Phantom Language Compiler. Also Java to Phantom translator, and preliminary code for Python to Phantom translator.

pfsextract Tool to extract persistent memory image from Phantom disk image file.

build Built executables for tools, scripts.

CHAPTER 3

Reference

3.1 Native API Reference

Native Phantom applications API Reference - set of classes available to user program written for Phantom OS.

Most of kernel features are available to Phantom program with the help of one of classes from `.internal` package.

Usually it is supposed that user code creates instance of `.internal` class and uses it to access kernel API. Sometimes there are handy wrappers from `.phantom` package.

3.1.1 Package .internal

.internal.string

It is just the same as any string constant in user program. You can use it's methods with string constant:

```
"abcd".substring( 3, 1 );
```

String encoding in Phantom is UTF-8. There will be corresponding `.internal.wstring` class for UTF-32 representation.

Most of the methods are self explaining.

```
int equals(var object);
.internal.string substring( var index : int, var length : int );
.internal.string concat( var s : string );
int length();
int strstr( var s : string );
int  toInt();
long toLong();
```

The next method parses JSON and returns tree of corresponding arrays/directories/leaf objects.

```
.internal.object parseJson();
```

This method returns not char, but byte at given position and will be renamed accordingly.

```
int charAt( var index : int );
```

.internal.stringbuilder

Mutable string optimized for being extended from the tail.

```
int equals(var object)
.internal.string toString()
.internal.string substring( var index : int, var length : int )
int byteAt( var index : int )
int concat( var s : string )
int length()
int strstr( var s : string )
```

The main difference from .internal.string is that `concat` does not return new string, but extends `this` object.

.internal.binary

Binary data buffer.

```
int  getByte( var index : int );
void setByte( var index : int, var value : int );
void setRange( var from : .internal.binary, var toPos : int, var fromPos : int, var
↪ln : int );
```

.internal.bitmap

Graphical image. Used for painting backgrounds and icons in UI code.

Parse PPM (P6) bitmap file contents. Alpha set to 255.

```
void loadFromString(var src : string);
```

There is a way to import images into the Phantom program:

```
bmpw = new .internal.bitmap();
bmpw.loadFromString(import "../resources/backgrounds/weather_sun_sm.ppm");
```

Here `import "file name"` operator creates a string constant which contains all the contents of the named file on the computer where we compile the program.

The `bmpw.loadFromString()` method converts file contents into an internal bitmap representation and then garbage collector disposes the string which contained file.

Phantom's persistence then keeps contents of bitmap object as long as you need it.

.internal.bootstrap

This class is used only during OS initialization and is not to be used by application code.

.internal.class

Note that `class` is reserved word in Phantom language, so if you have to refer to class class, it is done like this: `.internal."class"`.

Object of this class represents Phantom class. It can be used to check object type.

.internal.cond

Cond syncronization primitive.

Correspondingly, wait for event, wait with timeout, broadcast (wake up all waiters), and signal - wake up one waiting thread.

```
void wait( var toUnlock : .internal.mutex );
void twait( var toUnlock : .internal.mutex, var waitTimeMsec : int );
void broadcast();
void signal();
```

.internal.mutex

Mutex synchronization primitive. Take or release mutex.

```
void lock();
void unlock();

...
mtx.lock();
// update complex state
mtx.unlock();
...
```

.internal.directory

Hash map.

```
int put( var key : .internal.string, var value );
.internal.object get( var key );
int remove( var key : .internal.string );
int size();
```

Numeric classes

Classes `.internal.int`, `.internal.long`, `.internal.float` and `.internal.double` represent numeric types.

The only notable method now is `toString()`.

.internal.io

Represents file/device access.

```
void    open( var filename : string );
void    close();
void    read( var nBytes : int );
void    write( var data : string );
void    seek( var pos: int, var whence: int );
// ioctl
void    setAttribute( var name : string, var value : string );
string  getAttribute( var name : string );
```

.internal.stat

Access kernel statistics. `getStat()` returns statistics information, first arg is counter number and second is kind of statistics counter value.

Corresponding counter number values can be found in `<kernel/stats.h>`. There are no Phantom language defines yet, sorry.

Kind values are:

- 0: Current counter value
- 1: Average value per second
- 2: Value for this kernel boot
- 3: Value for previous boot
- 4: Value for the whole history of this OS instance

Note that 4-th kind, among others, lets you find out number of reboots done and snapshots made.

```
class .internal.stat
{
    int getStat( var nCounter : int, var kind : int );
    int getIdle();
};
```

.internal.tcp

This class will provide access to TCP protocol. Currently it can be used to execute HTTP connection and get a result.

```
class .internal.tcp
{
    // Call URL and return http server reply
    .internal.string curl( var host : .internal.string, var headers : .internal.
↪string ) [24] {  }
};
```

.internal.thread

Internal representation of thread. User code is not supposed to create objects of this type. It is possible but meaningless. Though it is possible to access such an object for own (current) thread.

```
class .internal.thread
{
    .internal.string toString();

    /**
     *
     * Return (possibly specific for this thread) OS
     * interface object, which is used to access public
     * OS services.
     *
     * @See .phantom.osimpl
     *
    **/
    .phantom.osimpl getOsInterface();

    /**
     *
     * Return owner of this thread. For the root OS
     * threads this can be null.
     *
    **/
    .phantom.user getUser();

    /**
     *
     * Return this thread's environment (Unix-style).
     *
    **/
    .phantom.environment getEnvironment();
};
```

.internal.time

Time related kernel services.

```
class .internal.time
{
    // classic Unix 32 bit time. TODO make it to be 64?
    .internal.int unixTime();

    void sleepSec( var timeSec : int );
    void sleepMsec( var timeMsec : int );

    // void runLater( var start : .phantom.runnable, var timeMsec : int );
};
```

.internal.udp

UDP protocol. Port numbers and IP addresses are in host byte order (ordinary integers).

```
class .internal.udp
{
    // Set local port number
    .internal.string bind( var port : .internal.int );
```

(continues on next page)

(continued from previous page)

```
    .internal.string recvFrom( var addr : .internal.int, var port : .internal.int );
    .internal.int sendTo( var data : .internal.string, var addr : .internal.int, var␣
↪port : .internal.int );
};
```

.internal.window

Access GUI subsystem.

```
class window
{
    int     getXSize();
    int     getYSize();

    int     getX();
    int     getY(); // Get window position on screen

    void    update(); // Bring window image to screen

    // Update window after each paint - flickers a bit.
    // Enabled by default. Turn off and update manually.
    void    setAutoUpdate( var auto : int );

    void    clear();
    void    fill(var fg : int);

    void    setBg( var bg : int ); // background color
    void    setFg( var fg : int ); // foreground color

    void    drawString( var x : int, var y : int, var s : string );
    void    drawImage( var x : int, var y : int, var img : .internal.object ); //␣
↪param is bitmap

    // You can put bitmap with basic drawImage and redraw part with this call
    void    drawImagePart( var x : int, var y : int, var img : .internal.object, var␣
↪xstart : int, var ystart : int, var xsize : int, var ysize : int ); // param is␣
↪bitmap

    //void    setWinSize( var xsize : int, var ysize : int );
    void    setWinPosition( var x : int, var y : int );

    void    drawLine( var x : int, var y : int, var xsize : int, var ysize : int );
    void    drawBox( var x : int, var y : int, var xsize : int, var ysize : int );

    void    fillBox( var x : int, var y : int, var xsize : int, var ysize : int );
    void    fillEllipse( var x : int, var y : int, var xsize : int, var ysize : int );

    //void    setEventHandler( var handler : .ru.dz.phantom.handler );
    void    setTitle( var title : string );

    // errno
    int     scrollHor( var x : int, var y : int, var xsize : int, var ysize : int,␣
↪var steps : int );
};
```

.internal.world

Access misc OS services which can't be accessed through `.internal` classes.

```
class .internal.world
{
    .internal.thread getMyThread();

    // returns errno
    int startThread( var entry : .phantom.runnable, var arg : .internal.object );

    // Print to system logging facility - practically it is a
    // log window in real OS and stdout in pvm_*.exe
    void log( var msg : .internal.string );
};
```

3.1.2 Package .phantom

Classes in `.phantom` package are not implemented in kernel but nevertheless provide basic OS functions too.

.phantom.application

This is a base class for Phantom OS application program. Instances of this class are application instances - documents, for example.

```
class .dz.clock extends .phantom.application
{
  .internal.bitmap appIcon;

  void clock() // Constructor
  {
    appIcon = new .internal.bitmap();
    appIcon.loadFromString(import "../resources/icons/clock_icon.ppm");
  }

  void run( var arg )
  {
    // Instance of clock is created and thread is run here
    // Create application window, paint clock, etc
  }

  .internal.string getDescription( )
  {
    return "Clock";
  }

  .internal.bitmap getIcon()
  {
    return appIcon;
  }
}
```

.phantom.runnable

Abstract base class for various callbacks and code that can be run in a thread.

```
class runnable
{
    // They call us here
    void run( var arg @const ) [8]
    {
        throw "Abstract runnable started";
    }
};
```

.phantom.environment

Similar to Unix process environment - `key=value` settings storage. Currently not really used.

```
class environment
{
    int set( var key: .internal.string, var val  : .internal.object );
    .internal.object get(var key: .internal.string );
};
```

3.2 Kernel API Reference

This section is addressed to Phantom OS kernel developer.

3.2.1 Threads and syncronization

Sync primitives:

- Spinlock
- Mutex
- Cond
- Semaphore

Spinlocks

```
#include <spinlock.h>

void        hal_spin_init(hal_spinlock_t *sl);
void        hal_spin_lock(hal_spinlock_t *sl);
void        hal_spin_unlock(hal_spinlock_t *sl);
```

Spinlock must be taken with interrupts disabled and for very short time, like few code lines. Special version of spinlock functions disables and restores interrupts.

```
void    hal_spin_lock_cli(hal_spinlock_t *sl);
void    hal_spin_unlock_sti(hal_spinlock_t *sl);
```

Spinlock can not be placed in paged/persistent memory, because it is forbidden to switch thread context with spinlock locked. If you absolutely have to place spinlock into a persistent memory object, use these functions:

```
// Turns off interrupts too
void    hal_wired_spin_lock(hal_spinlock_t *l);
void    hal_wired_spin_unlock(hal_spinlock_t *l);
```

Note that they wire (prevent pageout) memory page(s) spinlock resides in. It means that locking such a spinlock can cause pagein, which is disk IO and can take unexpectedly long time.

Note that there is a special `pvm_spinlock`, which is quite a different kind of spinlock and is dedicated to use with bytecode virtual machine.

Mutex

```
#include <kernel/mutex.h>

errno_t hal_mutex_init(hal_mutex_t *m, const char *name);
errno_t hal_mutex_lock(hal_mutex_t *m);
errno_t hal_mutex_unlock(hal_mutex_t *m);
errno_t hal_mutex_destroy(hal_mutex_t *m);
int     hal_mutex_is_locked(hal_mutex_t *m);

#define ASSERT_LOCKED_MUTEX(m) assert(hal_mutex_is_locked(m))
```

Important: Be careful using `hal_mutex_is_locked()`. Races possible - mutex can get locked by other thread just after return. The only safe use is for ASSERT_LOCKED_MUTEX.

Cond

```
#include <kernel/cond.h>

errno_t hal_cond_init( hal_cond_t *c, const char *name );
errno_t hal_cond_wait( hal_cond_t *c, hal_mutex_t *m );
errno_t hal_cond_timedwait( hal_cond_t *c, hal_mutex_t *m, long msecTimeout );
errno_t hal_cond_signal( hal_cond_t *c );
errno_t hal_cond_broadcast( hal_cond_t *c );
errno_t hal_cond_destroy( hal_cond_t *c );
```

Semaphore

The main reason for semaphores to exist when we have mutex/cond is that semaphore can be signalled (released) from interrupt and if you release it before waitng thread will wait (attempt to acquire) for it, it will not be put asleep.

```
int     hal_sem_init( hal_sem_t *s, const char *name );
void    hal_sem_release( hal_sem_t *s );
int     hal_sem_acquire( hal_sem_t *s );
errno_t sem_get_count(hal_sem_t *s, int *count);
void    hal_sem_destroy( hal_sem_t *s );
errno_t hal_sem_zero( hal_sem_t *s );
```

hal_sem_acquire Attempt to acquire a semaphore. Try to decrement semaphore value. If it becomes less than zero - sleep until semaphore is released (incremented) enough times for it to be decremented to zero.

hal_sem_release Increment semaphore value. If some thread attempted to acquire and sleeping in that state waiting for semaphore to become positive, wake up that thread.

sem_get_count Return current semaphore count.

hal_sem_zero Set semaphore to zero. Next hal_sem_acquire will sleep waiting for someone to release.

3.2.2 Allocators and Pools

Kernel allocators:

- Main physical memory allocator
- Address space allocator
- P+V allocator
- Kernel heap
- Pools

Physical memory allocator

Allocate and free physical computer memory. Allocation unit is page (usually 4094 bytes). Physical memory is usually quite limited resource, don't waste it.

> **Warning:** Allocated physical memory is not mapped into the kernel (or any other) address space and can not be accessed just after being allocated. You need to map it to some address in virtual address space, or use for physical IO.

```
errno_t hal_alloc_phys_page(physaddr_t *result);
void    hal_free_phys_page(physaddr_t page);
errno_t hal_alloc_phys_pages(physaddr_t *result, int npages);
void    hal_free_phys_pages(physaddr_t page, int npages);
```

Note: Before being freed physical memory must be umapped - detouched from address space.

If you need physical memory to be allocated and mapped to some address, you can use following functions:

```
void hal_pv_alloc( physaddr_t *pa, void **va, int size_bytes );
void hal_pv_free( physaddr_t pa, void *va, int size_bytes );
```

hal_pv_alloc Allocate physmem, allocate address space for it, and map memory to that address space. Panics if out of anything. Returns physical address of allocated memory in `pa`, virtual address in `va`, size is increased to be whole number of pages.

hal_pv_free Unmap, free address space and physical memory.

This set of fuctions is good if you need just big amount of kernel memory.

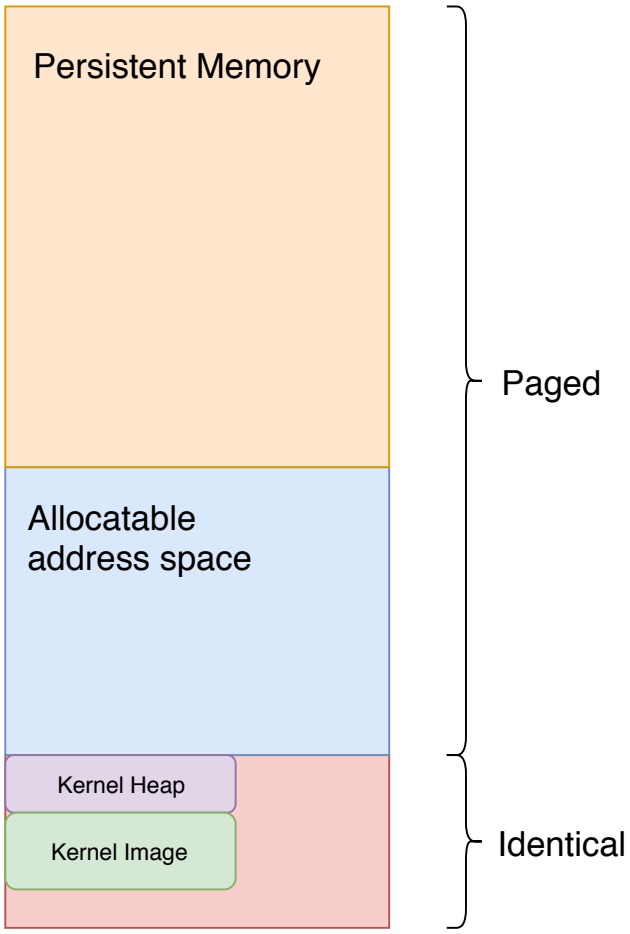

Fig. 1: Virtual memory layout for x86 architecture
Lowest part of memory is identically mapped - virtual address is equal to physical RAM address. It is not a requirement, it's just the easiest way to start the kernel. Blue part is address space that is available for allocations. Upper half is where applications live. Also referred to as object land.
Note that on other architectures (namely, on MIPS) address space structure is different.

Warning: If you need buffer to communicate with hardware, do not use these functions. Allocate and map memory manually, because you will need special mode of mapping.

Address space allocator

Allocate address space - page address or set of page addresses. This set of calls **does not allocate memory**, just interval of addresses to be mapped to some physical memory.

Typical use is to map device memory into the kernel address space.

```
errno_t hal_alloc_vaddress(void **result, int n_pages);
void hal_free_vaddress(void *addr, int n_pages);
```

Mapping and unmapping memory

This set of fuctions controls relations between physical memory and virtual address space. You can call these functions just for address space allocated with `hal_alloc_vaddress`, other parts of address space are conrolled by kernel in a special way.

```
void hal_page_control( physaddr_t pa, void *va, page_mapped_t mapped, page_access_t␣
↪access );
void hal_pages_control( physaddr_t pa, void *va, int n_pages, page_mapped_t mapped,␣
↪page_access_t access );
void hal_page_control_etc(
     physaddr_t pa, void *va,
     page_mapped_t mapped, page_access_t access,
     u_int32_t flags
    );
void hal_pages_control_etc( physaddr_t pa, void *va, int n_pages, page_mapped_t␣
↪mapped, page_access_t access, u_int32_t flags );
```

hal_pages_control_etc Map or unmap (depending on `mapped` parameter value) physical memory at `pa` to address space at `va`.

Possible values for `mapped` parameter are: - page_unmap: Unmap page(s) - page_map: Map page of RAM (cached access) - page_map_io: Map page of device memory (non-cached access)

Possible values for `access` parameter: - page_noaccess: No access (but mapping is set up in pagemap) - page_readonly: Read only access. - page_readwrite: Read and write access.

Physical memory access

Sometimes it is needed to copy data to or from physical memory. Set of functions helps to do it easily.

```
void hal_copy_page_v2p( physaddr_t to, void *from );
void memcpy_p2v( void *to, physaddr_t from, size_t size );
void memcpy_v2p( physaddr_t to, void *from, size_t size );
void memzero_page_v2p( physaddr_t to );
```

Kernel heap

Not much to say, use classic `malloc`, `calloc` and `free` functions as usual. Note that size of heap is fixed and you can easily run out of it. Use `hal_pv_alloc` if you need a lot of memory.

Pools

Aside from allocators, but used together is pools subsystem. Pool is an engine which can keep data structures (referred as *elements*) of one type inside. Structures can be accessed by handle. Pool keeps reference count of structure usage and can free it on reference count to become zero. Constructor and destructor functions can be registered if needed.

```
#include <kernel/pool.h>

pool_t *create_pool();
errno_t destroy_pool(pool_t *);
pool_t *create_pool_ext( int initial_elems, int arena_size );
```

After creating pool you can set additional properties.

flag_autoclean If pool itself is destroyed, destroy all of contents.

flag_autodestroy If element reference count becomes zero, destroy element.

flag_nofail If anything goes wrong, call `panic()`, never return an error. Don't use this mode.

init This function is called on element creation. An `arg` parameter of `pool_create_el` is passed to init, and its result is stored in pool. This function is, usually, allocates memory and sets up initial values for element.

If init function is not set up for pool, `arg` parameter of `pool_create_el` is stored in pool as is.

destroy Called on element destruction. Usually releases related (referenced by pool element) resources and deallocates element's memory.

```
pool_handle_t pool_create_el( pool_t *pool, void *arg );
void *pool_get_el( pool_t *pool, pool_handle_t handle );
errno_t pool_release_el( pool_t *pool, pool_handle_t handle );
errno_t pool_destroy_el( pool_t *pool, pool_handle_t handle );
```

pool_create_el Create new element in pool. An `arg` is poiner to structure to be placed in pool as is, or, if `init` (constructor) function is set for pool, an argument for that function. Reference count for new element is 1. Element handle is returned, or `INVALID_POOL_HANDLE` in case of error.

pool_destroy_el Destroy pool element with given handle, even if reference count for it is not zero.

pool_get_el Get pointer to pool element for given handle, increase reference count.

pool_release_el Tell pool that we do not use poiner to element any more. Decrease element count. Element can be destroyed if count becomes zero.

Here is a complex element creation scenario example. Element init function allocates new element and sets its name from parameter passed to `pool_create_el`. If element gets unused `f_destroy` deallocates it.

```
struct pe {
   char *name;
};

void *f_init( void *arg )
{
    struct pe *new_el = malloc( sizeof(struct pe) );
    new_el->name = strdup( arg );
}

void f_destroy( void *pool_el )
{
    struct pe *el = pool_el;
    free( pe->name );
```

(continues on next page)

(continued from previous page)

```
    free( pe );
}

...
pool = create_pool();
pool->flag_autoclean = 1;
pool->flag_autodestroy = 1;
pool->init = f_init;
pool->destroy = f_destroy;
...

pool_handle_t h = pool_create_el( pool, "new element name" );
```

Example of element access code. Caller passes us handle for a structure, we extract structure (incrementing its reference count), do an operation and release it, decrementing its reference count. This way we make sure that pool element still exists and will exist for the time we access it.

```
void w_control_set_visible( window_handle_t w, control_handle_t ch, int visible )
{
    control_t *cc = pool_get_el( w->controls, ch );
    if( !cc )
    {
        LOG_ERROR0( 1, "can't get control" );
        return;
    }

    cc->flags |= CONTROL_FLAG_DISABLED;
    if( visible ) cc->flags &= ~CONTROL_FLAG_DISABLED;

    w_paint_control( w, cc );

    pool_release_el( w->controls, ch );
}
```

Iterate through pool elements.

```
errno_t pool_foreach( pool_t *pool,
    errno_t (*ff)(pool_t *pool, void *el, pool_handle_t handle, void *arg),
    void *arg );

errno_t do_pool_forone( pool_t *pool,
    pool_handle_t handle,
    errno_t (*ff)(pool_t *pool, void *el, void *arg),
    void *arg );
```

pool_foreach Call `ff` for each element in pool, passing pool pointer, element pointer element handle and passed `arg`. Pool element refcount is incremented for the time `ff` is called and decremented after. If refcount becomes zero element is deleted.

If `ff` returns not zero, `pool_foreach` stops and returns the same value.

do_pool_forone Similar to `pool_foreach`, can be used to call `ff` just for one element. Does not destroy pool el in any case, even if refcount becomes zero.

Code example:

```
static errno_t do_paint_changed_control(pool_t *pool, void *el, pool_handle_t handle,␣
↪void *arg)
{
    control_t *cc = el;
    struct foreach_control_param *env = arg;
    paint_changed_control( env->w, cc);
    return 0;
}

void w_paint_changed_controls(window_handle_t w)
{
    struct foreach_control_param env;
    env.w = w;

    pool_foreach( w->controls, do_paint_changed_control, &env );
}
```

3.2.3 Interrupts

Hardware interrupts

```
errno_t hal_irq_alloc( int irq, void (*func)(void *arg), void *arg, int is_shareable␣
↪);
void hal_irq_free( int irq, void (*func)(void *arg), void *arg );
```

hal_irq_alloc Set interrupt handler.

Software interrupts

Software interrupts are called from hardware interrupt handler after all hardware interrupts are handled and interrupt state is left. Main use is thread preemption.

```
void hal_request_softirq( int sirq );
void hal_set_softirq_handler( int sirq, void (*func)(void *), void *_arg );

int hal_alloc_softirq(void); // Return next unused softirq number

void hal_enable_softirq(void);
void hal_disable_softirq(void);
```

3.2.4 Time and timers

Sleep functions

```
void hal_sleep_msec( int miliseconds );
void phantom_spinwait_msec( int miliseconds );
void tenmicrosec(void);
```

hal_sleep_msec Sleep for given time by switching current thread off the CPU. It does not guarantee that exact time will pass, for it can take some more time to return thread to CPU.

phantom_spinwait_msec Sleep by spinning - does not switch threads. You should be disabling interrupts for CPU not to be stolen by timer interrupt. Do not use if possible.

tenmicrosec Spin for ~10uSec. Also - disable interrupts.

Note: It is forbidden for a kernel code to sleep or wait for a long time keeping access to a persistent virtual memory.

Polled timeouts

These functions can be used to check if given time is passed.

```
typedef bigtime_t polled_timeout_t;

void set_polled_timeout( polled_timeout_t *timer, bigtime_t timeout_uSec );
bool check_polled_timeout( polled_timeout_t *timer );
```

set_polled_timeout Set up a timeout.

check_polled_timeout Check if requested time is passed.

```
polled_timeout_t tmo;
// start IO
set_polled_timeout( &tmo, CONST_IO_TIME*2 );
// Do something
if( check_polled_timeout( &tmo ) ) return -1; // Failed
```

Main system timer

```
#include <time.h>

bigtime_t hal_system_time(void); // uptime
bigtime_t hal_local_time(void);  // real time/date

time_t    time(time_t *timer);
void      set_time(time_t time);

// Fast, but less accurate time, sec
time_t    fast_time(void);

// Uptime in seconds
time_t    uptime(void);
```

bigtime_t Time in microseconds.

time_t Traditional Unix time in seconds.

Time of day

Time of day subsystem keeps track of current time and date.

```
time_t mktime(struct tm *);
size_t strftime(char * , size_t, const char * , const struct tm * );
char *ctime_r(const time_t *, char *);
```

(continues on next page)

(continued from previous page)

```
struct tm *gmtime_r(const time_t *, struct tm *);
struct tm *localtime_rb(bigtime_t timer, struct tm *tmb);
```

In-interrupt timer calls

It is possible to call some callback after a given time right from a timer interrupt. Should be used very carefuly.

```
#include <kernel/timedcall.h>

typedef void (*timedcall_func_t)( void * );

typedef struct timedcall
{
    timedcall_func_t        f;
    void                *arg;
    long                msecLater;

    hal_spinlock_t *    lockp;

    // Private filds follow

} timedcall_t;
```

f Function to be called by timer.

arg Parameter to pass to `f`.

msecLater Time to pass before `f` is called.

```
#define TIMEDCALL_FLAG_PERIODIC         (1 << 0)
#define TIMEDCALL_FLAG_CHECKLOCK        (1 << 2)

void phantom_request_timed_call( timedcall_t *entry, u_int32_t flags );
void phantom_undo_timed_call(timedcall_t *entry);
```

phantom_request_timed_call Request call of function after given time is passed. All the parameters are to be set up in `timedcall_t` structure.

phantom_undo_timed_call Un-request previously requested timed call. Note that just when this function is called timed call may start to execute. So call to it does not guarantee that call did not happen. See `TIMEDCALL_FLAG_CHECKLOCK` for interlocking.

TIMEDCALL_FLAG_PERIODIC Execute timed call periodically.

TIMEDCALL_FLAG_CHECKLOCK If this flag is given, timed call won't happen as long as spinlock given in `lockp` field is locked. This way you can guarantee that timed call is not performed when it is not needed anymore and you are going to disable it with `phantom_undo_timed_call`.

In-thread timer calls

Kernel call can request for a callback in a thread context to be done after a given time. Such a callback should not take long for it will prevent other callbacks to be executed.

```
#include <kernel/net_timer.h>
```

(continues on next page)

(continued from previous page)

```
int set_net_timer(net_timer_event *e, unsigned int delay_ms,
    net_timer_callback callback, void *args, int flags);
int cancel_net_timer(net_timer_event *e);
```

set_net_timer Request call to `callback` after `delay_ms`.

cancel_net_timer Cancel request. Request still can be fired even after call to this function.

3.2.5 Threads

Phantom kernel supports kernel multithreading and quite powerful multitasking features in general.

```
#include <threads.h>

typedef struct phantom_thread phantom_thread_t;

tid_t   hal_start_thread(void (*thread)(void *arg), void *arg, int flags);
errno_t t_kill_thread( tid_t tid );
```

Thread properties

```
tid_t   get_current_tid(void);

errno_t t_set_owner( tid_t tid, void *owner );
errno_t t_get_owner( tid_t tid, void **owner );

errno_t t_new_ctty( tid_t tid );
errno_t t_get_ctty( tid_t tid, struct wtty ** );

errno_t t_set_pid( tid_t tid, pid_t pid );
errno_t t_get_pid( tid_t tid, pid_t *pid );

errno_t t_set_priority( tid_t tid, int prio );
errno_t t_get_priority( tid_t tid, int *prio );

errno_t t_add_flags( tid_t tid, u_int32_t set_flags );
errno_t t_remove_flags( tid_t tid, u_int32_t reset_flags );
errno_t t_get_flags( tid_t tid, u_int32_t *flags );
```

t_set_owner Is not interpreted by threads engine, used to connect low level thread to bytecode virtual machine threads.

t_new_ctty Detouch thread's controlling terminal from parent thread's tty. It is up to the caller to connect controlling terminal somewhere.

t_set_pid Used by POSIX subsystem only. Native Phantom code does not use concept of processes.

Thread priorities from 1 to 0xF are normal timesharing priorities. Priority of 0 is idle time execution only. If prio is above `THREAD_PRIO_MOD_REALTIME` (0x10 to 0x1F) - thread is real time, and will take all CPU preventing any lower prio threads to run. Use with big care.

Following functions control current thread only.

```
errno_t         t_current_set_priority( int prio );
errno_t         t_current_get_priority( int *prio);
```

(continues on next page)

(continued from previous page)

```
errno_t         t_current_set_name(const char *name);

errno_t         t_current_set_death_handler(void (*handler)( phantom_thread_t * ));
```

Thread flags:

- THREAD_FLAG_USER: Runs in user mode. Not implemented.
- THREAD_FLAG_VM: Runs bytecode virtual machine thread, owner points to VM thread object.
- THREAD_FLAG_JIT: JITted VM thread. Not imlpemented.
- THREAD_FLAG_TIMEDOUT: Cond (or something else) was timed out.
- THREAD_FLAG_UNDEAD: This thread can't be killed for some reason. Usually it's some special one like CPU idle thread.
- THREAD_FLAG_NOSCHEDULE: Must not be selected by scheduler in usual way - per CPU 'parking' (idlest) thread
- THREAD_FLAG_SNAPPER: I am a snapper thread.
- THREAD_FLAG_HAS_PORT: This thread (possibly) owns port (see newos/ports)

The only flag that can be set with t_add_flags/t_remove_flags is `THREAD_FLAG_HAS_PORT`.

DPC: Thread pools

DPC (Deferred Procedure Call) requests are supposed to be used in interrupts handlers to offload long part of interrupt processing to thread and not to keep separate thread for every need of every driver.

DPC subsystem keeps pool of threads used to handle requests, and adds threads to pool if all of existing threads are used.

```
#include <kernel/dpc.h>

void dpc_request_init(dpc_request *rq, void (*_func)(void *));
void dpc_request_trigger(dpc_request *me, void *_arg);
```

DPC can be triggered from interrupt and does not require any memory allocation or other resources at trigger point.

Example code:

```
dpc_request drq;

static void driver_init( void )
{
    dpc_request_init( &drq, &dpc_handler );
}

static void dpc_handler( void *arg )
{
    // Read data from device
}

static void interrupt_handler( void *arg )
{
    if( read_request )
```

(continues on next page)

(continued from previous page)

```
        dpc_request_trigger( &drq, 0 );
}
```

Classic thread pool subsystem - can not be triggered from interrupt.

```
typedef void (*runnable_t)( void *arg );
void run_in_thread_pool( runnable_t func, void *arg );
```

run_in_thread_pool Requests `func` to be started with `arg` in separate thread.

3.2.6 Disk IO

Low level disk IO subsystem is targeted mostly to support paging function as main Phantom IO subsystem is persistent memory.

IO Request

```
#include <pager_io_req.h>

typedef struct pager_io_request
{
    physaddr_t     phys_page;               // physmem address
    disk_page_no_t disk_page;               // disk address in pages - as pager␣
↪requested (ignored by io code)

    unsigned char  flag_pagein;            // Read
    unsigned char  flag_pageout;           // Write

    void           (*pager_callback)(
        struct pager_io_request *req, int write );

    errno_t        rc;

    // Other fields are private

} pager_io_request;
```

phys_page Physical memory address for IO.

disk_page Disk page address for IO. In 4K pages.

flag_pagein If not zero - perform read.

flag_pageout If not zero - perform write.

pager_callback If not zero - call when request is done or failed.

rc Result code. Zero on success.

Syncronous disk IO

To be described.

Disk registration

Here is an example of disk driver entry point and registering a new disk in the system.

```
#include <disk_q.h>

phantom_disk_partition_t *p = phantom_create_virtio_partition_struct( cfg.capacity, &
↪vdev );
errno_t ret = phantom_register_disk_drive(p);

static errno_t memdisk_AsyncIo( struct phantom_disk_partition *part, pager_io_request␣
↪*rq )
{
    phantom_device_t *dev = part->specific;
    rq->rc = 0;

    size_t size = rq->nSect * part->block_size;
    off_t shift = rq->blockNo * part->block_size;

    if( size+shift > dev->iomemsize )
    {
        rq->rc = EIO;
        pager_io_request_done( rq );
        return EIO;
    }

    if(rq->flag_pageout)
        rq->rc = EIO;
    else
        // read
        memcpy_v2p( rq->phys_page, (void *)dev->iomem+shift, size );

    pager_io_request_done( rq );
    return rq->rc;
}

phantom_disk_partition_t *phantom_create_memdisk_partition_struct(
    phantom_device_t *dev, long size, int unit )
{
    phantom_disk_partition_t * ret =
        phantom_create_partition_struct( 0, 0, size );

    ret->asyncIo = memdisk_AsyncIo;
    ret->flags |= PART_FLAG_IS_WHOLE_DISK;

    ret->specific = dev;
    strlcpy( ret->name, "MEMD0", sizeof(ret->name) );
    ret->name[4] += unit;
    return ret;
}


static void memdisk_connect( phantom_device_t *dev, int nSect )
{
    phantom_disk_partition_t *part =
        phantom_create_memdisk_partition_struct( dev, nSect, 0 );
    if(part == 0)
    {
```

(continues on next page)

(continued from previous page)

```
        SHOW_ERROR0( 0, "Failed to create whole disk partition" );
        return;
    }

    errno_t err = phantom_register_disk_drive(part);
    if(err)
    {
        SHOW_ERROR( 0, "Disk %p err %d", dev, err );
        return;
    }
}
```

Now some explanation.

memdisk_AsyncIo This is driver entry point which is called with a new IO request and has to handle it. In this simple example request is processed right in place, but in real driver this function will, usually, just start IO and set things up so that pager_io_request_done() will be called in device interrupt.

pager_io_request_done This function must be called for a request when it is processed (with either success or error) with driver. Request `rc` field must be non-zero if request is not executed for some reason.

phantom_create_memdisk_partition_struct This is a driver's helper function which creates driver descriptor structure for a kernel. This structure is used by kernel to access this driver for IO. Note `ret->asyncIo = memdisk_AsyncIo;` line, which tells what function to call for incoming IO request for this driver's device.

phantom_register_disk_drive Driver init code calls this function to tell kernel that this is a disk that is served by this driver. Kernel will try to identify disk contents - if there are partitions on disk, kernel will add partition handling structures and in turn try to identify partition contents. Finally either disk or partition will be recognized as Phantom OS disk, some kind of file system (FAT32 is the only one that is supported), or not recognized at all.

Partitions

The only type of partitioning supported in current kernel is traditional PC MBR type one. Additional kinds of disk breakdown can be added easily.

File systems

Native Phantom code can be (and usually is) written so that it needs no access to file systems. But it is unrealistic yet and not reasonable to live completely without traditional FS access.

Connecting FS driver

Here is an example of how to connect a file system to kernel.

List of available FS drivers is in fs_map.c file. Each FS is defined with name, FS probe function and FS start function.

FS probe entry point is called for each whole disk or partition and for each registered FS:

```
errno_t fs_probe_fat(phantom_disk_partition_t *p )
{
    unsigned char buf[PAGE_SIZE];

    SHOW_FLOW( 1, "%s look for FAT", p->name );
```

(continues on next page)

(continued from previous page)

```
    switch( p->type )
    {
        case 1: // FAT 12
        case 4: // FAT 16 below 32M
        case 6: // FAT 16 over 32M
        // Check more types we support
        break;
    default:
        SHOW_ERROR( 1, "Not a FAT partition type 0x%X", p->type );
        return EINVAL;
    }

    if( phantom_sync_read_sector( p, buf, 0, 1 ) )
    {
        SHOW_ERROR( 0, "%s can't read sector 0", p->name );
        return EINVAL;
    }

    if( (buf[0x1FE] != 0x55) || (buf[0x1FF] != 0xAA) )
    {
        SHOW_ERROR0( 1, "No magic" );
        return EINVAL;
    }

// Do more checks, return zero if we decide this is our FS disk

return 0;
}
```

If we said that it is our partition, the next entry point will be called by kernel:

```
errno_t fs_start_fat( phantom_disk_partition_t *p )
{
    FATFS *fs = calloc( sizeof(FATFS), 1);

    fs->dev = p;

    // Do FS specific init code

    uufs_t *ufs = fatff_create_fs( fs );
    if( !ufs )
    {
        SHOW_ERROR( 0, "can't create uufs for %s", p->name );
        return ENOMEM;
    }

    char pname[FS_MAX_MOUNT_PATH];
    partGetName( p, pname, FS_MAX_MOUNT_PATH );

    char amnt_path[128];
    if( ufs && auto_mount( pname, ufs, amnt_path, sizeof(amnt_path), AUTO_MOUNT_FLAG_
↪AUTORUN ) )
        SHOW_ERROR( 0, "can't automount %s", p->name );

    return 0;
}
```

This code creates specific structure for use inside the FS driver (`FATFS`), creates uufs_t file system descriptor, and mounts new file system. Note that given in partition access structure `p` is stored in `fs->dev` so that FS can access its disk partition.

Here is how its done:

```
int disk_write( FATFS *fs, const void* data, int sector, int nsect)
{

    if( phantom_sync_write_sector( fs->dev, data, sector, nsect ) )
        return RES_ERROR;

    // Put to cache only if successfully written to disk
    if( fs->cache )
        cache_put_multiple( fs->cache, sector, nsect, data );

    return 0;
}

static int disk_read ( FATFS *fs, void* data, int sector, int nsect)
{
    if( fs->cache )
        if( 0 == cache_get_multiple( fs->cache, sector, nsect, data ) )
            return 0;

    if( phantom_sync_read_sector( fs->dev, data, sector, nsect ) )
        return RES_ERROR;

    if( fs->cache )
        cache_put_multiple( fs->cache, sector, nsect, data );

    return 0;
}
```

This code also shows how to use disk cache subsystem. See detailed explanation for it in *Disk cache subsystem* below.

Now lets find out how to setup file system descriptor to be used by kernel to access a file system. The `uufs` structure describes instance of file system. It has function pointers used to access this instance and pointer to implementation specific state.

```
struct uufs
{
    char *              name;
    void *              impl;

    errno_t             (*open)(struct uufile *, int create, int write);
    errno_t             (*close)(struct uufile *);

    uufile_t *          (*namei)(struct uufs *fs, const char *filename);

    errno_t             (*symlink)(struct uufs *fs, const char *src, const char *dst);
    errno_t             (*mkdir)(struct uufs *fs, const char *path);

    uufile_t *          (*root)(struct uufs *fs);

    errno_t             (*dismiss)(struct uufs *fs);
};
```

impl Private implementation specific state for this FS instance.

namei This function is called for a file name before any access to this file is possible. This function must create `uufile_t` structure which is used by kernel to access file.

open This function is used to open or create file. Called for uufile obtained from `namei` or `root`.

close File is not will be used any more. FS code can free associated resources.

root It is an analog of `namei` for the root directory of this file system.

mkdir Create a directory.

symlink Create a symlink.

dismiss This file system is not needed any more. Close everything, flush cache and release all resources.

```
struct uufile
{
    struct uufileops *  ops;
    size_t              pos;
    struct uufs *       fs;
    unsigned            flags;
    const char *        name;   // This entry's name, or zero if none
    void *              impl; // implementation specific

    int                 refcount; // n of refs to this node - TODO!
    hal_mutex_t         mutex; // serialize access
};

typedef struct uufile uufile_t;

#define UU_FILE_FLAG_DIR        (1<<0) // Dir
#define UU_FILE_FLAG_NET        (1<<1) // Socket
#define UU_FILE_FLAG_TCP        (1<<2) // Stream
#define UU_FILE_FLAG_UDP        (1<<3) // Dgram
#define UU_FILE_FLAG_MNT        (1<<4) // Mount point - FS root
#define UU_FILE_FLAG_PIPE       (1<<5) // Pipe
#define UU_FILE_FLAG_RDONLY     (1<<6) // Write op forbidden

//! Is open now, unlink/destroy_uufile will close first
#define UU_FILE_FLAG_OPEN       (1<< 8)

//! On destroy implementation must be freed
#define UU_FILE_FLAG_FREEIMPL   (1<< 9)

//! Do not destroy - it's a link to static instance
#define UU_FILE_FLAG_NODESTROY  (1<<10)
```

This structure describes any accessible node in file system, like file, directory, pipe. etc.

ops Pointer to a list of entry points for operations available. Note that each file can have its own set of operations. You can have specific ops for directory and regular file, for example.

pos Current position in file.

fs Our filesystem instance.

name File name.

impl Implementation specific private state.

refcount Number of references to this instance.

flags Kind of file and operation modes for it, see defines above.

mutex Mutex to take when working with this file.

Basic operations on generic `uufile_t`:

```
uufile_t *copy_uufile( uufile_t *in );
uufile_t *create_uufile(void);

void set_uufile_name( uufile_t *in, const char *name );

void link_uufile( uufile_t *in );
void unlink_uufile( uufile_t *in );
```

FS specific operations on files:

```
struct uufileops
{
    size_t  (*read)(    struct uufile *f, void *dest, size_t bytes);
    size_t  (*write)(   struct uufile *f, const void *src, size_t bytes);

    errno_t (*readdir)( struct uufile *f, struct dirent *dirp );

    errno_t (*seek)(    struct uufile *f );

    errno_t (*stat)(    struct uufile *f, struct stat *dest);
    int     (*ioctl)(   struct uufile *f, errno_t *err, int request, void *data, int␣
↪dlen );

    size_t  (*getpath)( struct uufile *f, void *dest, size_t bytes);

    ssize_t (*getsize)( struct uufile *f);
    errno_t (*setsize)( struct uufile *f, size_t size);

    errno_t (*chmod)( struct uufile *f, int mode);
    errno_t (*chown)( struct uufile *f, int user, int grp);

    errno_t (*unlink)( struct uufile *f );

    errno_t (*getproperty)( struct uufile *f, const char *pName, char *pValue, size_t␣
↪vlen );
    errno_t (*setproperty)( struct uufile *f, const char *pName, const char *pValue );
    errno_t (*listproperties)( struct uufile *f, int nProperty, char *pValue, size_t␣
↪vlen );

    // used when clone file
    void *  (*copyimpl)( void *impl);
};
```

read, write Access file data.

readdir Access directory contents. Return ENOENT if end of dir reached.

seek Notify fs driver that curr pos (uufile.pos) is changed, ask to validate. If returns nonzero, uufile.pos may be changed again.

stat Read file metadata.

ioctl Classic Unix ioctl.

getproperty, setproperty, listproperties Phantom's extended ioctl, name=value style. See *Device properties* for more info.

getpath Read file path.

getsize, setsize Read or control file size.

chmod Change file access rights, unix-style.

chown Change file owner, unix style.

unlink Unlink file, delete if it was last link.

copyimpl Create another instance of uufile impl part.

This structure is usually shared by all instances of uufile structure for given file system.

There is a ready made directory support structures and code. It can be used to keep in-memory directories structure either from RAMFS or for some procfs like file system.

This implemented is quite naive and uses linear lists of directory entries. If used for a big directories can be slow.

```
typedef struct dir_ent
{
    queue_chain_t       chain;

    const char *        name; // malloc'ed
    int                 flags;
    uufile_t *          uufile;
    void *              unused;
} dir_ent_t;

uufile_t *create_dir(void);

errno_t unlink_dir_name( uufile_t *dir, const char *name );
errno_t unlink_dir_ent( uufile_t *dir, uufile_t *deu );

uufile_t *lookup_dir( uufile_t *dir, const char *name, int create, uufile_t␣
↪*(*createf)() );

errno_t get_dir_entry_name( uufile_t *dir, int n, char *name );
int common_dir_read(struct uufile *f, void *dest, size_t bytes);

extern struct uufileops common_dir_fops;
```

create_dir Create uufile with support for directory operations. An `common_dir_fops` set of methods will be used. Following operations are also possible.

unlink_dir_name Unlink by name. Will delete uufile referenced by directory entry if it was last reference.

unlink_dir_ent Unlink by referenced uufile.

lookup_dir Find or create directory entry. If entry does not exist and `create` is not zero, `createf` will be called to create uufile to be referenced by entry.

get_dir_entry_name Return n-th entry's name.

common_dir_read General impl of read syscall for directory. Reads records structured as `struct dirent`.

Disk cache subsystem

Default disk IO cache machinery. Disk IO subsystem does not use cache subsystem automatically, you have to add it to your code manually. The reason is simple - Phantom persistent memory subsystem is itself a huge cache, and does not need separate IO cache at all. So adding cache to FS implementation must be done in FS code, not in disk IO code.

```
#include <kernel/disk_cache.h>

cache_t * cache_init( size_t block_size );

errno_t cache_get( cache_t *c, long blk, void *data );
errno_t cache_get_multiple( cache_t *c, long blk, int nblk, void *data );

errno_t cache_put( cache_t *c, long blk, const void *data );
errno_t cache_put_multiple( cache_t *c, long blk, int nblk, const void *data );

errno_t cache_set_writeback( cache_t *c, writeback_f_t *func, void *opaque );
errno_t cache_flush_all( cache_t *c );

errno_t cache_destroy( cache_t * );
```

cache_init Create cache of default size for given page (sector) size. Returns cache struct or null on fail (usually out of mem).

cache_get Find a cache entry and get data from it, or return ENOENT if corresponding disk block is not in cache.

cache_get_multiple Find a cache entry and get data from it, or return ENOENT - multisector.

cache_put Place data to cache - find or reuse entry as needed.

cache_put_multiple Place data to cache - find or reuse entry as needed - multisector.

cache_set_writeback Set function to call if cache needs to write out data from cache to disk.

cache_flush_all Make sure all the cached data is written to disk.

> **Warning:** It is not guaranteed that cache_flush_all returns after all outstanding data is surely written to disk.

cache_destroy Destroy cache.

3.2.7 Network

Phantom has classical TCP/IP network stack.

Interfaces

This chapter describes how to connect new network driver to a network interface.

Driver entry points for network IO:

```
#include <kernel/ethernet_defs.h>
#include <kernel/net.h>

int pcnet32_read( struct phantom_device *dev, void *buf, int len);
int pcnet32_write(struct phantom_device *dev, const void *buf, int len);
int pcnet32_get_address( struct phantom_device *dev, void *buf, int len);
```

pcnet32_get_address Read network card MAC address.

```
phantom_device_t * dev = malloc(sizeof(phantom_device_t));
dev->name = "pcnet";
dev->seq_number = seq_number++;
```

(continues on next page)

(continued from previous page)

```
dev->dops.read = pcnet32_read;
dev->dops.write = pcnet32_write;
dev->dops.get_address = pcnet32_get_address;

// Init other fields of dev as it is needed

ifnet *interface;
if( if_register_interface( IF_TYPE_ETHERNET, &interface, dev) )
    printf("Failed to register interface for %s", dev->name );
else
    if_simple_setup(interface, WIRED_ADDRESS, WIRED_NETMASK, WIRED_BROADCAST, WIRED_
↪NET, WIRED_ROUTER, DEF_ROUTE_ROUTER );
```

if_simple_setup Right now we use some predefined addresses first, but this function starts DHCP to get correct setup. It is a hack and must be fixed later.

```
#define WIRED_ADDRESS    IPV4_DOTADDR_TO_ADDR(10, 0, 2, 123)
#define WIRED_NETMASK    0xffffff00
#define WIRED_BROADCAST  IPV4_DOTADDR_TO_ADDR(10, 0, 2, 0xFF)
#define WIRED_NET        IPV4_DOTADDR_TO_ADDR(10, 0, 2, 0)
#define WIRED_ROUTER     IPV4_DOTADDR_TO_ADDR(10, 0, 2, 123)
#define DEF_ROUTE_ROUTER IPV4_DOTADDR_TO_ADDR(10, 0, 2, 2)
```

3.2.8 Main drivers subsystem

These are structures that represent a driver. Just fields that are of use for a driver are described here. (There are more fields in the structures we talk about below, but most of them are just for kernel.)

Driver entry points:

```
#include <device.h>

struct phantom_dev_ops
{
    int (*start)(struct phantom_device *dev);
    int (*stop)(struct phantom_device *dev);

    int (*read)(struct phantom_device *dev, void *buf, int len);
    int (*write)(struct phantom_device *dev, const void *buf, int len);

    int (*get_address)(struct phantom_device *dev, void *buf, int len);

    errno_t (*ioctl)(struct phantom_device *dev, int type, void *buf, int len);

    errno_t (*getproperty)( struct phantom_device *dev, const char *pName, char␣
↪*pValue, int vlen );
    errno_t (*setproperty)( struct phantom_device *dev, const char *pName, const char␣
↪*pValue );
    errno_t (*listproperties)( struct phantom_device *dev, int nProperty, char␣
↪*pValue, int vlen );
};
```

start Called if kernel is going to use this device - before first call to any other entry point.

stop Called if no access to driver is going to be any more.

read Read data from device. For character or network devices only. Block devices register themselves for block IO requests separately.

write Write data to device.

get_address Special for network devices - must return MAC address.

ioctl For POSIX subsystem. Classic ioctl.

getproperty Reach man's ioctl, get device's named property value. Can be used from POSIX and object land code.

getproperty Set device's named property value.

listproperties Tell names of properties device supports.

Device descriptor structure:

```
struct phantom_device
{
    const char *                name;
    int                         seq_number;
    void *                      drv_private;
    phantom_dev_ops_t           dops;

    addr_t                      iobase;
    int                         irq;
    physaddr_t                  iomem;
    size_t                      iomemsize;

    struct properties           *props;
};

typedef struct phantom_device phantom_device_t;

void devfs_register_dev( phantom_device_t* dev );
```

name Device name.

seq_number Sequence number of device, if driver presents more than one.

drv_private Pointer to driver's private data structure which is specific for this device.

dops Entry points, must be filled for kernel to access.

iobase, irq, iomem, iomemsize Can be used or ignored by driver. Kernel does not interpret.

props *Device properties*, see separate description.

Device properties

Here is an usage example for a simplest way to handle properties.

Driver init code, set entry points for properties:

```
dev->dops.getproperty = es1370_getproperty;
dev->dops.setproperty = es1370_setproperty;
dev->dops.listproperties = es1370_listproperties;
```

List available properties. This function will be called with increasing `nProperty` value until it returns an error.

```
static const char *pList = "sampleRate";

static errno_t es1370_listproperties( struct phantom_device *dev, int nProperty, char␣
↪*pValue, int vlen )
{
    if( nProperty > 0 ) return ENOENT;
    strlcpy( pValue, pList, vlen );
    return 0;
}
```

Get value of property:

```
static errno_t      es1370_getproperty( struct phantom_device *dev, const char *pName,
↪ char *pValue, int vlen )
{
    es1370_t *es = dev->drv_private;

    if(0 == stricmp(pName, "samplerate"))
    {
        snprintf( pValue, vlen, "%d", es->samplerate );
        return 0;
    }

    return ENOTTY;
}
```

```
static errno_t      es1370_setproperty( struct phantom_device *dev, const char *pName,
↪ const char *pValue )
{
    es1370_t *es = dev->drv_private;

    if(0 == stricmp(pName, "samplerate"))
    {
        if( 1 != sscanf( pValue, "%d", &es->samplerate ) )
            return EINVAL;
        set_sampling_rate(dev, es->samplerate);
        return 0;
    }
return ENOTTY;
}
```

There is a set of functions to support more complex implementation of properties machinery.

```
typedef enum
{
    pt_int32,
    pt_mstring,         // malloced string
    pt_enum32,              // enum int32 - unimpl!
} property_type_t;

struct property;

typedef struct properties {
    const char *          prefix;         // 4-byte char prefix of this group, like
↪'dev.', 'gen.' or 'fsp.'

    struct property     *list;
```

(continues on next page)

(continued from previous page)

```
    size_t              lsize;

    void *              (*valp)(struct properties *ps, void *context, size_t offset );
} properties_t;

typedef struct property {
    property_type_t type;
    const char      *name;
    size_t          offset;
    void            *valp;

    char            **val_list; // for enums

    void            (*activate)(struct properties *ps, void *context, size_t offset,␣
↪void *vp );
    errno_t         (*setf)(struct properties *ps, void *context, size_t offset, void␣
↪*vp, const char *val);
    errno_t         (*getf)(struct properties *ps, void *context, size_t offset, void␣
↪*vp, char *val, size_t len);
} property_t;

errno_t gen_dev_listproperties( struct phantom_device *dev, int nProperty, char␣
↪*pValue, int vlen );
errno_t     gen_dev_getproperty( struct phantom_device *dev, const char *pName, char␣
↪*pValue, int vlen );
errno_t     gen_dev_setproperty( struct phantom_device *dev, const char *pName, const␣
↪char *pValue );

errno_t gen_listproperties( properties_t *ps, int nProperty, char *pValue, int vlen );
errno_t     gen_getproperty( properties_t *ps, void *context, const char *pName, char␣
↪*pValue, int vlen );
errno_t     gen_setproperty( properties_t *ps, void *context, const char *pName,␣
↪const char *pValue );
```

To use this properties engine define `properties_t` structure for a driver, add array of property definitions (`property_t`) and use `gen_dev_listproperties`, ... as an access functions.

Device's `props` field must point to `properties_t` structure.

3.2.9 Video drivers subsystem

Video drivers provide access to the graphical hardware of the computer.

Video driver

Video driver must be, at least, able to:

- Turn on 24 or 32 bit video mode for the graphics hardware
- Set up linear video buffer access
- Provide functions to write and read video buffer

There are ready made functions to access usual frame buffer formats for 24 or 32 bit buffers.

Video driver descriptor structure:

```
struct drv_video_screen_t
{
    const char  *name;

    int     xsize;
    int     ysize;
    int     bpp;

    int     mouse_x;
    int     mouse_y;
    int     mouse_flags;

    char *  screen;

    int     (*probe) (void); // Returns true if hardware present, sets xsize/ysize.
    int     (*start) (void); // Start driver, switch to graphics mode.
    errno_t (*accel) (void); // Start driver in accelerating mode - video mode is␣
↪already set by VESA, just add some acceleration to existing drv
    int     (*stop)  (void); // Stop driver, switch to text mode. Can be called in␣
↪unstable kernel state, keep it simple.

    void    (*update) (void);

    void    (*bitblt) (const rgba_t *from, int xpos, int ypos, int xsize, int ysize,␣
↪zbuf_t zpos, u_int32_t flags);
    void    (*winblt) ( const window_handle_t from, int xpos, int ypos, zbuf_t zpos);
    void    (*readblt) ( rgba_t *to, int xpos, int ypos, int xsize, int ysize);

    void    (*mouse)  (void); // mouse activity detected - callback from driver

    void    (*mouse_redraw_cursor)(void);
    void    (*mouse_set_cursor)(drv_video_bitmap_t *cursor);
    void    (*mouse_disable)(void);
    void    (*mouse_enable)(void);

// Acceleration

    void    (*copy) (int src_xpos, int src_ypos, int dst_xpos, int dst_ypos, int␣
↪xsize, int ysize );
    void    (*clear) (int xpos, int ypos, int xsize, int ysize );
    void    (*bitblt_part) (const rgba_t *from, int src_xsize, int src_ysize, int src_
↪xpos, int src_ypos, int dst_xpos, int dst_ypos, int xsize, int ysize, zbuf_t zpos,␣
↪u_int32_t flags );
};

#define VIDEO_PROBE_FAIL    0
#define VIDEO_PROBE_SUCCESS 1
#define VIDEO_PROBE_ACCEL   2
```

probe This is the entry point which is called by graphics subsystem on start. Provided function must find out if there is some video device that this driver can control. Return values are: `VIDEO_PROBE_FAIL` - no hardware found, driver is skipped. `VIDEO_PROBE_SUCCESS` - driver can work with this hardware, either in basic or accelerated mode (if accelerated - driver must provide corresponding entry points). `VIDEO_PROBE_ACCEL` - driver can not initialize hardware and setup frame buffer, but can provide additional acceleration. The last mode is usually assumes that basic VESA driver will init hardware and this driver will do just acceleration. If driver reports success, its descriptor structure must be filled with maximum possible screen size and bpp.

start Start driver. Graphics subsystem decided that this driver is the best among all and want it to control the display.

Driver must set up video mode, set entry points and map frame buffer into the kernel address space.

stop Turn graphics off, set character mode if possible, release resources.

accel Start in accelerating mode. Other driver is already set up video mode, this one has just to provide acceleration entry points.

Following entry points are called after driver started to access device:

update Finally after frame buffer is updated. Used if hardware needs some kick to redisplay screen or if screen is virtual or network.

bitblt Copy bitmap to screen.

winblt Copy contents of a window to screen.

readblt Read bitmap from screen. Used by default mouse cursor engine.

Accelerating driver

Video driver which can not work by itself and relies on VESA or some other video driver to set up video mode and access to frame buffer, but can provide accelerated functions.

Entry points:

mouse_set_cursor In accelerated mode supposed to provide hardware cursor.

mouse_redraw_cursor New cursor coordinated are in `mouse_x` and `mouse_y` fields. Move hardware cursor to that position.

copy Copy (bitblt) part of screen to other place.

clear Fill part of screen with color.

3.2.10 Graphics

Phantom graphical subsystem is built in 4 layers:

- Video driver
- Bitblt to screen functions - copy window image to screen supporting z-order of windows
- Windowing system: windows, painting to windows, events, window decorations and related support
- Controls - graphical components supporting user interaction: buttons, check boxes, text entry fields, etc.

Windows

```
#include <video/window.h>

window_handle_t drv_video_window_create(int xsize, int ysize, int x, int y, rgba_t bg,
↪ const char* title, int flags );

void    drv_video_window_destroy(window_handle_t w);

void    w_to_top(window_handle_t w);
void    w_to_bottom(window_handle_t w);

void    w_clear( window_handle_t win );
void    w_fill( window_handle_t win, rgba_t color );
```

(continues on next page)

(continued from previous page)

```
void    w_draw_rect( window_handle_t win, rgba_t color, rect_t r );
void    w_fill_rect( window_handle_t win, rgba_t color, rect_t r );

void    w_draw_pixel( window_handle_t w, int x, int y, rgba_t color );
void    w_draw_line( window_handle_t w, int x1, int y1, int x2, int y2, rgba_t c);
void    w_fill_ellipse( window_handle_t w, int x,int y,int lx,int ly, rgba_t c);
void    w_fill_box( window_handle_t w, int x,int y,int lx,int ly, rgba_t c);
void    w_draw_box( window_handle_t w, int x,int y,int lx,int ly, rgba_t c);
void    w_draw_bitmap( window_handle_t w, int x, int y, drv_video_bitmap_t *bmp );

// Draw with alpha blending
void    w_draw_blend_bitmap( drv_video_window_t *w, int x, int y, drv_video_bitmap_t␣
↪*bmp );

void    w_move( window_handle_t w, int x, int y );

errno_t w_scroll_hor( window_handle_t w, int x, int y, int xs, int ys, int s );
void    w_scroll_up( window_handle_t win, int npix, rgba_t color);

void    w_set_title( window_handle_t w,  const char *title );
void    w_get_bounds( window_handle_t w, rect_t *out );
void    w_set_visible( window_handle_t h, int v );
void    w_set_bg_color( window_handle_t w, rgba_t color );
```

Simple example of using windows:

```
drv_video_window_t *w = drv_video_window_create(
    WXS, WYS, 300, 300,
    COLOR_BLACK, "Test Window", WFLAG_WIN_DECORATED );

w_draw_line( w, 0, 0, WXS, WYS, COLOR_RED );
w_fill_ellipse( w, 30, 30, 15, 27, COLOR_BLUE );

w_font_draw_string( w, &drv_video_8x16san_font, "Bitmap font",
    COLOR_BLACK, COLOR_GREEN, 0, 0 );

w_fill_box( w,  40, 32, 33, 10, COLOR_RED );

font_handle_t font = w_get_system_font_ext( 50 );
w_ttfont_draw_string( w, font,
    "TrueType text", COLOR_LIGHTRED,
    10, 50 );

w_release_tt_font( font );
drv_video_winblt( w );
```

Events

Events are messages sent to window system requesting update its state. Most of events are processed by window system itself. Some event are supposed to be processed by window owning code.

```
#include <event.h>

struct ui_event
```

(continues on next page)

(continued from previous page)

```
{
    queue_chain_t             echain;

    int                       type; // UI_EVENT_TYPE_
    int                       extra;

    time_t                    time;

    int                       abs_x;
    int                       abs_y;
    int                       abs_z; // z of clicked win

    int                       rel_x;
    int                       rel_y;
    int                       rel_z;

    struct drv_video_window * focus;

// Shift, alt, etc - actual for key and mouse events
    int                       modifiers;

    union {
        struct {
            int               vk;
            int               ch;
        } k;

        struct {
            int               buttons;
            int               clicked;
            int               released;
        } m;

        // WIN and GLOBAL events
        struct {
            int               info;   // UI_EVENT_WIN_
            rect_t            rect;
            rect_t            rect2;
        } w;
    };
};

typedef struct ui_event ui_event_t;
```

There are four types of events:

- UI_EVENT_TYPE_MOUSE - Mouse move, click or release
- UI_EVENT_TYPE_KEY - Key press or release. Fields specific for key events grouped in `k` part of union.
- UI_EVENT_TYPE_WIN - Window state change request. Relevant fields are in the `w` part of union.
- UI_EVENT_TYPE_GLOBAL - Global events are usually converted to multiple WIN events. Specific data is in the `w` part.

Following fields are possibly used with any type of event.

- `time` - event creation time.
- `abs_x`, `abs_y`, `abs_z` - absolute (screen) coordinates, associated with event.

- rel_x, rel_y, rel_z - relative (window) coordinates, associated with event.
- modifiers - shift/alt/ctrl/etc modifiers actual for both mouse and key events.
- focus - target window for this event.
- extra - additional information or event. If event is related to the control (like button or menu item), control *id* is placed here.

If type equals UI_EVENT_TYPE_MOUSE, this is a mouse event. Following parameters are relevant in this case:

- buttons - each bit corresponds to mouse button pressed or released. Bit masks UI_MOUSE_BTN_LEFT, UI_MOUSE_BTN_RIGHT and UI_MOUSE_BTN_MIDDLE can be used to check for mouse button pressed state.
- clicked - bits are set just for mouse click (transition to pressed state).
- released - bits are set just for release.

If buttons == 0 then event is mouse hover. If clicked and released are zero but buttons is not - it is a drag event.

If type equals UI_EVENT_TYPE_KEY, this is a keyboard event. Information about the key that is changed is in k.ch field. This field contains printable character or functional key code. Key codes for functional keys are defined in compat/uos/keyboard.h. You can check for key to be functional with

KEY_IS_FUNC(k.ch) macros.

Note that there are key events for both key press and release. Differ release from press by modifiers field, which will have UI_MODIFIER_KEYUP bit set for release event. See event.h for other modifiers. Most useful ones are UI_MODIFIER_SHIFT, UI_MODIFIER_CTRL and UI_MODIFIER_ALT.

There are macros to test modifiers state for usual cases.

- UI_MOD_DOWN() - Key is down and no control/alt/shift is pressed
- UI_MOD_CTRL_DOWN() - Key is down and just control is pressed
- UI_MOD_ALT_DOWN() - Key is down and just alt is pressed

If type equals UI_EVENT_TYPE_WIN, this is a window event.

Window events are mostly internal windows system machinery events, but sometimes must be generated or processed by user code.

- UI_EVENT_WIN_GOT_FOCUS - sent to window that just became focused.
- UI_EVENT_WIN_LOST_FOCUS - sent to window that just lost focus.
- UI_EVENT_WIN_DESTROYED - sent to window that is being destroyed.
- UI_EVENT_WIN_REDECORATE - internal machinery event, causes window decorations (title, borders) to be repaint.
- UI_EVENT_WIN_REPAINT - internal machinery event, sent to request window to be repaint.
- UI_EVENT_WIN_BUTTON_ON - state of the control added to this window is changed. New state is *ON*.
- UI_EVENT_WIN_BUTTON_OFF - state of the control added to this window is changed. New state is *OFF*.
- UI_EVENT_WIN_TO_TOP - internal machinery event. Request to bring window to top position.
- UI_EVENT_WIN_TO_BOTTOM - internal machinery event. Request to bring window to bottom position.
- UI_EVENT_WIN_MOVE - internal machinery event. Request to bring window to new x/y.

If `type` equals `UI_EVENT_TYPE_GLOBAL`, this is a global event.

Global events are converted to multiple window events for related windows.

- `UI_EVENT_GLOBAL_REPAINT_RECT` - repaint part of all windows which intersect with given rectangle.

Functions to send events

```
void ev_q_put_mouse( int x, int y, int buttons );
void ev_q_put_key( int vkey, int ch, int modifiers );
void ev_q_put_win( int x, int y, int info, struct drv_video_window *focus );
void ev_q_put_global( ui_event_t *e );
void ev_q_put_any( ui_event_t *ie );
```

ev_q_put_mouse Put mouse event onto the main event q.

ev_q_put_key Put key event onto the main event q. An `vkey` parameter value is ignored.

ev_q_put_win Put window event onto the main event q.

ev_q_put_global Put global event onto the main event q. Supposed that event system will decide which windows have to receive this event.

ev_q_put_any Put any type of event onto the main event q. BE CAREFUL. NB - don't forget to set focus before sending events to window!

This function is not supposed to be used outside of events system.

How to process events in kernel code

There is a function pointer in window structure which points to the function called to process events for this window.

```
int (*inKernelEventProcess)( struct drv_video_window *w, struct ui_event *e );
```

On window creation default event processor pointer is stored here. If you want to override, call default processor function (`defaultWindowEventProcessor`) for all of the events before or after special processing. If, for some reason, you will replace event processor not for main, but for title window (that is an additional window which is used to paint title bar), pass event to `w_titleWindowEventProcessor`.

Window and control focus

One of the visible windows is usually *focused* - technically it means that this window will receive keyboard events. Focused window is usually the topmost, but not necessarily - window is getting focus if mouse is over this window, but is brought to top only if clicked to.

You can select next window to be focused with Ctrl-TAB or Alt-TAB key sequence, hovering mouse over or clicking with mouse.

If window has controls in it (buttons, switches, etc), one of these controls is also focused. This control will get all the keyboard input sent to this window.

Next control can be selected to be in focus by TAB key press.

Keyboard maps

There is basic support for national keyboards in Phantom. Currently it has one hardcoded keymap, but it is not a big problem to add other keymaps and keymap selection machinery. Please leave an issue in project GitHub if you need specific keymap.

Controls

Control is graphical element attached to window and processing user input.

Create control:

```
#include <video/control.h>

control_handle_t w_add_button( window_handle_t w, int id, int x, int y, drv_video_
↪bitmap_t *bmp, drv_video_bitmap_t *pressed, int flags );
control_handle_t w_add_radio_button( window_handle_t w, int id, int group_id, int x,␣
↪int y );
control_handle_t w_add_checkbox( window_handle_t w, int x, int y );
control_handle_t w_add_menu_item( window_handle_t w, int id, int x, int y, int xsize,␣
↪const char*text, color_t text_color );
control_handle_t w_add_label( window_handle_t w, int x, int y, int xsize, int ysize,␣
↪const char *text, color_t text_color );
control_handle_t w_add_label_transparent( window_handle_t w, int x, int y, int xsize,␣
↪int ysize, const char *text, color_t text_color );
control_handle_t w_add_label_ext( window_handle_t w, int x, int y, int xsize, int␣
↪ysize, const char *text, color_t text_color,
                                drv_video_bitmap_t *bg, uint32_t flags );
control_handle_t w_add_text_field( window_handle_t w, int x, int y, int xsize, int␣
↪ysize, const char *text, color_t text_color );
control_handle_t w_add_scrollbar_ext( window_handle_t w, int x, int y, int xsize, int␣
↪ysize, int minval, int maxval, uint32_t flags );
control_handle_t w_add_scrollbar( window_handle_t w, int x, int y, int xsize, int␣
↪ysize, int maxval );
```

Change control state or parameters. Text functions accept UTF-8 encoding.

```
void w_control_set_text( window_handle_t w, control_handle_t c, const char *text,␣
↪color_t text_color );
void w_control_get_text( window_handle_t w, control_handle_t c, char *text_buf, size_
↪t buf_size );
void w_control_set_icon( window_handle_t w, control_handle_t ch, drv_video_bitmap_t␣
↪*icon );

//! NB! Allocates new bitmaps and does alpha blending with basic window background
void w_control_set_background( window_handle_t w, control_handle_t ch,
    drv_video_bitmap_t *normal, drv_video_bitmap_t *pressed, drv_video_bitmap_t␣
↪*hover  );

void w_control_set_visible( window_handle_t w, control_handle_t ch, int visible ); //␣
↪unimpl yet
void w_control_set_position( window_handle_t w, control_handle_t ch, int x, int y );
```

Set flags for a control:

```
void w_control_set_flags( window_handle_t w, control_handle_t ch, int toSet, int␣
↪toReset );
```

Following flags are defined:

CONTROL_FLAG_DISABLED This control is disabled - not painted and does not process input.

CONTROL_FLAG_NOPAINT Control is not painted, but processes input. Use if you paint control manually.

CONTROL_FLAG_NOBORDER Do not paint border around the control. Border still will be paint if control is in focus.

CONTROL_FLAG_CALLBACK_HOVER Callback function will be called on mouse over the control.

CONTROL_FLAG_CALLBACK_KEY Call callback on any key press. If this flag is not set, callback is just called for enter in text entry field or key that changes on/off state for buttons and similar controls.

CONTROL_FLAG_TOGGLE Button or menu item switches on and off with each press.

CONTROL_FLAG_HORIZONTAL Lay control horizontally - not yet used.

CONTROL_FLAG_TEXT_RIGHT Put button text to the right of image.

CONTROL_FLAG_NOFOCUS This control is passive - no events, no focus.

CONTROL_FLAG_NOBACKGROUND Do not paint control background - Unused?

CONTROL_FLAG_READONLY Control is just for display, don't accept user input - TODO implement

CONTROL_FLAG_ALT_FG Control foreground (active part) is paint in a different way. Supported by scroll bar only.

CONTROL_FLAG_ALT_BG Control background (passive part) is paint in a different way Supported by scroll bar only.

Set callback to be called on major control state change.

```
void w_control_set_callback( window_handle_t w, control_handle_t c, control_callback_
↪t cb, void *callback_arg );
```

Set control's children - controlled objects. If just window is given, switch its visibility. If window and control - switch control visibility.

```
void w_control_set_children( window_handle_t w, control_handle_t c, window_handle_t w_
↪child, control_handle_t c_child );
```

Show bullet with number (`count` parameter) in top right corner

```
void w_control_set_notify( window_handle_t w, control_handle_t ch, int count );
```

Set context (right click) menu for the control. Note that it is possible to set context menu for a whole window too.

```
void w_control_set_menu( window_handle_t w, control_handle_t ch, window_handle_t m );
```

For switch type controls (button, check box, etc) - set or get on/off (pressed/released) state of control.

```
void w_control_set_state( window_handle_t w, control_handle_t ch, int pressed );
void w_control_get_state( window_handle_t w, control_handle_t ch, int *ret );
```

Control scrollbar position and size. - value - current value. Determines start position of scroll bar. - width - width of scroll bar - how big part of possible values it takes. If == maxval - minval - takes 100% of scrollbar size

Set value or width to be negative to disable display of bar at all.

```
void w_control_set_value( window_handle_t w, control_handle_t ch, int value, int␣
↪width );  //< For scrollbar - set value & bar handle width
void w_control_get_value( window_handle_t w, control_handle_t ch, int *value, int␣
↪*width ); //< For scrollbar - get value & bar handle width
```

Fonts

Phantom graphics subsystem supports bitmap and TrueType fonts.

```
#include <video/font.h>
```

Preinstalled set of bitmap fonts:

drv_video_16x16_font, drv_video_8x16ant_font, drv_video_8x16bro_font, drv_video_8x16cou_font, drv_video_8x16med_font, drv_video_8x16rom_font, drv_video_8x16san_font, drv_video_8x16scr_font, drv_video_kolibri1_font, drv_video_kolibri2_font, drv_video_gallant12x22_font, drv_video_freebsd_font.

Window functions to work with bitmap fonts:

```
void w_font_draw_string(
    window_handle_t win,
    const drv_video_font_t *font,
    const char *s,
    const rgba_t color,
    const rgba_t bg,
    int x, int y );

void w_font_scroll_line(
    window_handle_t win,
    const drv_video_font_t *font, rgba_t color );

void w_font_tty_string(
    drv_video_window_t *win,
    const struct drv_video_font_t *font,
    const char *s,
    const rgba_t fg_color,
    const rgba_t bg_color,
    int *x, int *y );
```

w_font_draw_string Paint string with given font.

w_font_scroll_line Scroll window one line up using given font height.

w_font_tty_string Paint string processing TTY control chars: \n, \r, some of ANSI control sequences (text color). Update given coordinates.

Truetype fonts support lets caller choose font style and size and provides painting and size request functions.

Access font:

```
font_handle_t w_get_system_font_ext( int font_size );
font_handle_t w_get_system_font( void );

font_handle_t w_get_system_mono_font_ext( int font_size );
font_handle_t w_get_system_mono_font( void );

font_handle_t w_get_tt_font_file( const char *file_name, int size );
font_handle_t w_get_tt_font_mem( void *mem_font, size_t mem_font_size, const char*␣
↪diag_font_name, int font_size );
```

(continues on next page)

(continued from previous page)

```
errno_t w_release_tt_font( font_handle_t font );
```

w_get_system_font_ext Get handle of default system font for given font size.

w_get_system_mono_font_ext Get handle of default monospaced (tty style) font for given size.

w_get_tt_font_file Load font from file - requires some filesystem to be running.

w_get_tt_font_mem Load font from memory - usually from persistent binary object.

w_release_tt_font Release font when it is not needed any more.

Use font:

```
void w_ttfont_draw_string(
    window_handle_t win,
    font_handle_t font,
    const char *s, const rgba_t color,
    int x, int y );

void w_ttfont_draw_string_ext(
    window_handle_t win,
    font_handle_t font,
    const char *str, size_t strLen,
    const rgba_t color,
    int win_x, int win_y,
    int *find_x, int find_for_char );


void w_ttfont_draw_char(
    window_handle_t win,
    font_handle_t font,
    const char *str, const rgba_t color,
    int win_x, int win_y );

void w_ttfont_string_size( font_handle_t font,
    const char *str, size_t strLen,
    rect_t *r );
```

w_ttfont_draw_string Paint string with given ttf font.

w_ttfont_draw_string_ext Extended string paint version. Returns X coordinate of `find_for_char` character in `find_x` variable. This feature is good for painting cursor line.

w_ttfont_draw_char Paint one character with TrueType font.

Note: It is not a good idea to paint text character by character, for kerning is not processed if you do it this way.

---

w_ttfont_string_size Calculate bounding rectangle for string.

There are UTF-32 versions of these functions exist, see header file.

UTF-8 and UTF-32

Main string encoding in Phantom is UTF-8. But there are parts of system (namely - keyboard driver) that expect UTF-32 encoding. Conversion is supported, of course.

Rectangle algebra

There is a set of functions to work with window/image coordinates.

They are based on a rect_t rectangle descriptor:

```
#include <video/rect.h>

typedef struct rect
{
    int x, y;
    int xsize, ysize;
} rect_t;

#define rect_copy( out, in ) rect_add( out, in, in )

void rect_add( rect_t *out, rect_t *a, rect_t *b );
int rect_mul( rect_t *out, rect_t *a, rect_t *b );
int rect_sub( rect_t *out1, rect_t *out2, rect_t *old, rect_t *new );

int rect_eq( rect_t *a, rect_t *b );
int rect_empty( rect_t *a );

int rect_includes( rect_t *a, rect_t *b );
int rect_intersects( rect_t *a, rect_t *b );
int point_in_rect( int x, int y, rect_t *r );

int rect_dump( rect_t *a );
```

rect_add Resulting rectangle is rectangle that includes both of source ones.

rect_mul Resulting rectangle is intersection. Function returns true if intersection exists.

rect_sub Calculates two rectangles which together cover space which is covered by `old`, but not covered by `new`. Returns nonzero if out2 is not equal to out1 and not empty.

rect_eq True if rectangles are equal.

rect_empty True if at least one of rectangle sizes is negative or zero.

rect_includes Returns true if a includes (or is equal to) b.

rect_intersects Returns true if a intersects with (or is equal to) b.

point_in_rect Returns true if point belongs to rectangle.

3.2.11 Persistent virtual memory

This is the core of native Phantom personality.

Persistent memory is used by Phantom OS bytecode vitrual machine (referred to as pvm). This memory is occupied by pvm objects and is subject of garbage collector work.

Kernel can access this memory and create and read objects in there, but there are some rules.

First of all, this memory is snapshotted to disk from time to time, and it must be in consistent state during snapshot. It means that when key stage if snapshot is going on no thread can access this memory.

Snapshots are fast

Note that key stage of snapshot is extremely short, so it does not stop any activity for a noticeable time.

There are two functions that are used to interlock access to persistent memory.

```
#include <kernel/snap_sync.h>

void vm_lock_persistent_memory( void );
void vm_unlock_persistent_memory( void );
```

vm_lock_persistent_memory Request access to persistent memory address space, prevent snapshots.

vm_unlock_persistent_memory Release access to persistent memory address space, enable snapshots.

Kernel code must call `vm_lock_persistent_memory` before it is going to access persistent memory and `vm_unlock_persistent_memory` as soon as possible after it.

Warning: Keeping access to persistent memory for too long will keep other threads waiting for snapshot to finish for too long too, because snapshot can not happen as long as you keep persistent memory locked. Be really quick.

Objects and classes

Phantom VM is an object-oriented VM (actually it has special support for functional programming too). Everything inside persistent memory is object. Object is instance of class. Which is object too.

Regular objects contain just pointers to other objects. Irregular (internal classes) objects contain some kind of data. For exmaple, object of class `internal.string` contains text string.

Strings are files

Actually phantom strings do not restrict its contents in any way. String can contain just any binary data including binary zeroes. That is why strings in Phantom are frequently used as... anonymous files. It is handy to put assets into strings and Phantom language compiler has special support for this kind of use. For example, some applications keep bitmap pictures in strings. See an example for *.internal.bitmap* class.

Internal classes

Phantom userland (virtual bytecode machine) code communicates with OS kernel by means of so called *internal* classes. Each internal class is implemented by in-kernel C code.

Source code of internal classes

Please refer to *Package .internal* for a complete reference of existing internal classes.

Objects as they look from C

From the kernel point of view PVM object is just a structure with an array or structure at the end. If it is a regular object, it's data part is array of pointers to other objects. If it is of internal class, object contents is a structure of some type.

```
#include <vm/object.h>

struct object_PVM_ALLOC_Header
{
    unsigned int                object_start_marker;
    volatile int32_t            refCount;
    unsigned char               alloc_flags;
    unsigned char               gc_flags;
    unsigned int                exact_size;
};

struct pvm_object_storage
{
    struct object_PVM_ALLOC_Header _ah;

    struct pvm_object_storage *    _class;
    struct pvm_object_storage *    _satellites;
    u_int32_t                      _flags;
    unsigned int                   _da_size;

    unsigned char                  da[];
};

typedef struct pvm_object_storage   pvm_object_storage_t;
typedef struct pvm_object_storage * pvm_object_t;
```

_class Pointer to an object which describes class of this object.

_satellites Used in a very special cases where there some additional data must be associated with our object. Examples are weak references (object must keep list of weak references that look at it) or Java-style syncronization machinery, which needs mutex to be associated with arbitrary object.

_flags Used to denote special cases or help to check for some types of objects very fast. Example is object of `.internal.int` class. It would be too long to go through the class reference to check if this object is an int.

_da_size Size of object contents **in bytes**.

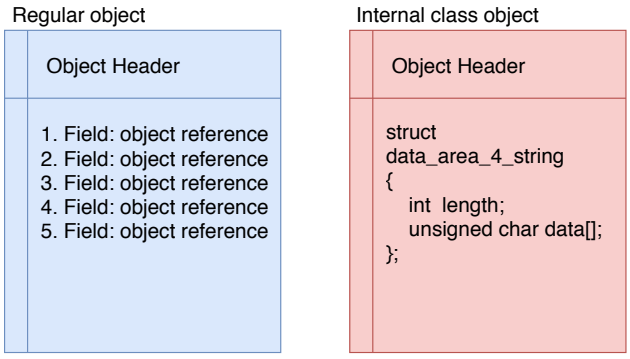

Fig. 2: Layout of regular and `.internal` class object
Regular objects can contain references (pointers) to other objects only. Internal ones can have any content which is interpreted by in-kernel class implementation.

There are numerous functions to access objects in different ways.

```
pvm_object_t pvm_create_object(pvm_object_t type);
pvm_object_t pvm_get_class( pvm_object_t o );

#define      pvm_is_null( o ) ...
#define      pvm_isnull( o ) pvm_is_null( o )

pvm_object_t pvm_get_field( pvm_object_t , unsigned int no );
void         pvm_set_field( pvm_object_t , unsigned int no, pvm_object_t value );
```

pvm_create_object Create object of given class. For most internal classes there is a `pvm_get_NAME_class(void)` function exist. For example, `pvm_get_window_class(void)` for `.internal.window`. So if you need a new int, `pvm_create_object( pvm_get_int_class() )` will do.

pvm_get_class Find out class of object.

pvm_is_null Check if object is null.

pvm_get_field, pvm_set_field Access fields of regular objects.

```
int pvm_object_class_exactly_is( pvm_object_t object, pvm_object_t tclass );
int pvm_object_class_is_or_parent( pvm_object_t object, pvm_object_t tclass );
int pvm_object_class_is_or_child( pvm_object_t object, pvm_object_t tclass );
```

Check object type.

pvm_object_class_exactly_is Find if object is of exactly this class. This one is rarely needed.

pvm_object_class_is_or_parent True if object class is `tclass` or its parent.

pvm_object_class_is_or_child True if object class is `tclass` or its child. This is the most typical type check.

Arrays operations. Note that there is no syncronization provided by basic array operations. You have to provide your own protection.

```
pvm_object_t pvm_get_array_ofield(pvm_object_t o, unsigned int slot  );
void         pvm_set_array_ofield(pvm_object_t o, unsigned int slot, pvm_object_t␣
↪value );

int          pvm_get_array_size(pvm_object_t array);
void         pvm_append_array(pvm_object_t array, pvm_object_t value_to_append );
void         pvm_pop_array(pvm_object_t array, pvm_object_t value_to_pop );
```

pvm_get_array_ofield Read field of array.

pvm_set_array_ofield Set array field. Will silently extend array if you access absent element.

pvm_get_array_size How many slots in array right now.

pvm_append_array Extend array by one slot and put object to last slot. Practically push to stack. **Mind the syncronization!**

pvm_pop_array Return last element, decreasing size of array by one. Pop from stack.

Debugging and printing.

```
void         pvm_object_print( pvm_object_t );
void         pvm_object_dump( pvm_object_t o );
void         dumpo( addr_t addr );
```

(continues on next page)

(continued from previous page)

```
void         pvm_puts(pvm_object_t o );
pvm_object_t pvm_get_class_name( pvm_object_t );
```

Running PVM code

Run new VM instance in a new thread. An `object` parameter must be of `.phantom.runnable` class.

```
errno_t pvm_run_new_thread( pvm_object_t object, pvm_object_t arg );
```

This function will create new thread and execute method (with ordinal of 8) with given object as *this* and passing `arg` to that method.

3.2.12 Unix subsystem

Phantom OS includes simple POSIX/Linux emulation.

(This part of book is not complete)

File IO

BeOS ports

This is not POSIX, but is more or less meaningless without POSIX. Ports is a microkernel style gear that can be used to write server processes that serve other parts of OS. Port is named message passing IPC channel. Port message is message type (integer) and message body, which is simply a byte buffer. Message type is not interpreted by kernel. Its meaning is defined by server party.

```
#include <newos/port.h>

errno_t phantom_port_create(port_id *ret, int32 queue_length, const char *name);
errno_t phantom_port_close(port_id id);
errno_t phantom_port_delete(port_id id);
errno_t phantom_port_find(port_id *ret, const char *port_name);
errno_t phantom_port_buffer_size(ssize_t *sizep, port_id port);
errno_t phantom_port_buffer_size_etc(ssize_t *sizep, port_id port,
                                     uint32 flags,
                                     bigtime_t timeout);
errno_t phantom_port_count(int32 *countp, port_id port);
errno_t phantom_port_read(ssize_t *len,
                                     port_id port,
                                     int32 *msg_code,
                                     void *msg_buffer,
                                     size_t buffer_size);
errno_t phantom_port_read_etc(ssize_t *len,
                                     port_id port,
                                     int32 *msg_code,
                                     void *msg_buffer,
                                     size_t buffer_size,
                                     uint32 flags,
                                     bigtime_t timeout);
errno_t phantom_port_write(port_id port,
                                     int32 msg_code,
```

(continues on next page)

(continued from previous page)

```
                                    void *msg_buffer,
                                    size_t buffer_size);
errno_t phantom_port_write_etc(port_id port,
                                    int32 msg_code,
                                    void *msg_buffer,
                                    size_t buffer_size,
                                    uint32 flags,
                                    bigtime_t timeout);
```

phantom_port_create Create port win given `name`. The `queue_length` parameter sets number of outstanding messages that can wait in port queue for processing.

phantom_port_find Find port created by other party by name.

phantom_port_buffer_size_etc Get size of buffer required to receive next message from port.

phantom_port_buffer_size Get size of buffer required to receive next message from port. Wait forever.

phantom_port_write_etc Send a message to `port`. Message type is in `msg_code` parameter. Message itself is in the `msg_buffer`, its size is `buffer_size`. If `flags` has value of `PORT_FLAG_TIMEOUT`, function will fail after `timeout` microseconds if unable to send message.

phantom_port_read_etc Get next message from port. Parameters are similar to `phantom_port_write_etc`.

3.2.13 Kernel infrastructure services

Misc info

Current architecture and board (hw conf) names.

```
#include <kernel/init.h>

extern char arch_name[];
extern char board_name[];
```

Kernel subsystems init and stop

Any kernel source file can have own init and stop code. Use following macros:

```
#include <kernel/init.h>

#define INIT_ME(__init1,__init2,__init3)
#define STOP_ME(__stop1,__stop2,__stop3)
```

Usage:

```
static void init_module( void )
{
...
}

INIT_ME(0,init_module,0);
```

Three parameters of `INIT_ME` macros are functions (or zero to skip) that will be called at different moments.

An `__init1` is called very early, most of the kernel services are not ready at this moment. Use just for modules that will be needed for later starting code. No threads, no mutex/cond/etc, no network, no disks.

An `__init2` is called when most of the kernel is working.

An `__init3` is called very late. Quite everything is running at this moment, object land is available. Use for slow and not critical components or additional startup of things that started in basic form before.

Chained buffers - cbuf

Can be used to carry data of unknown length between kernel modules. Widely used in network stack. There are tools to concatenate cbufs, cut off data at front or at tail, such as protocol headers and sufixes. Cbufs can also be extended at start or end to add protocol headers or some data at end.

Special interface exist to create and free cbufs in interrupt context.

```
#include <newos/cbuf.h>

cbuf *cbuf_get_chain(size_t len);
void cbuf_free_chain(cbuf *buf);

cbuf *cbuf_get_chain_noblock(size_t len);
void cbuf_free_chain_noblock(cbuf *buf);

size_t cbuf_get_len(cbuf *buf);
int cbuf_is_contig_region(cbuf *buf, size_t start, size_t end);
void *cbuf_get_ptr(cbuf *buf, size_t offset);

int cbuf_memcpy_to_chain(cbuf *chain, size_t offset, const void *_src, size_t len);
int cbuf_memcpy_from_chain(void *dest, cbuf *chain, size_t offset, size_t len);

cbuf *cbuf_merge_chains(cbuf *chain1, cbuf *chain2);
cbuf *cbuf_duplicate_chain(cbuf *chain, size_t offset, size_t len, size_t leading_
↪space);

cbuf *cbuf_truncate_head(cbuf *chain, size_t trunc_bytes, bool free_unused);
int cbuf_truncate_tail(cbuf *chain, size_t trunc_bytes, bool free_unused);

int cbuf_extend_head(cbuf **chain, size_t extend_bytes);
int cbuf_extend_tail(cbuf *chain, size_t extend_bytes);
```

cbuf_get_chain, cbuf_get_chain_noblock Get chain of given size. Noblock version can be used in interrupt.

cbuf_free_chain, cbuf_free_chain_noblock Release chain that is not needed any more.

cbuf_get_len Find number of bytes stored in chain.

cbuf_is_contig_region Check if some region of stored data is in one chain buffer, and is not broken in parts.

cbuf_get_ptr Get pointer to specific part of buffer. Make sure that part you need is in one chain buffer with `cbuf_is_contig_region`.

cbuf_memcpy_to_chain Copy from source buffer to chain. Given `offset` is starting position in chain to copy to.

cbuf_memcpy_from_chain Copy from `offset` position in chain to destination buffer.

cbuf_merge_chains Build new chain by concatenation of two given chains.

cbuf_duplicate_chain Create a copy of chain.

cbuf_truncate_head

cbuf_truncate_tail

cbuf_extend_head

cbuf_extend_tail

Hash tables - khash

Kernel hash tables are used around to map keys (like names or ports, or IP addressed) to handles or structure pointers.

```
#include <kernel/khash.h>

void *hash_init(unsigned int table_size, int next_ptr_offset,
    int compare_func(void *a, const void *key),
        unsigned int hash_func(void *a, const void *key, unsigned int range)
    );

int hash_uninit(void *_hash_table);

int hash_insert(void *_hash_table, void *_elem);
int hash_remove(void *_hash_table, void *_elem);

void *hash_find(void *_hash_table, void *e);
void *hash_lookup(void *_hash_table, const void *key);

struct hash_iterator *hash_open(void *_hash_table, struct hash_iterator *i);
void hash_close(void *_hash_table, struct hash_iterator *i, bool free_iterator);
void *hash_next(void *_hash_table, struct hash_iterator *i);
void hash_rewind(void *_hash_table, struct hash_iterator *i);

void hash_dump(void *_hash_table);

unsigned int hash_hash_str( const char *str );
```

hash_init Set up a new hash. An `table_size` parameter determines size of hash table. An `next_ptr_offset` parameter gives offset to a field in a structure which is used as pointer to next sibling in hash node list. Use `offsetof(type_name, next_ptr)` and add `void *next_ptr` field in a structure that you're going to keep in hash.

Hash function should calculate hash on either e or key, depending on which one is not NULL.

```
unsigned int hash_func(void *e, const void *key, unsigned int range);
```

Compare function should compare the element with the key, returning 0 if equal, other if not.

```
int compare_func(void *e, const void *key);
```

Note: Compare func can be null, in which case the hash code will compare the key pointer with the target.

Usage example. Create hash table for network interfaces. Note `next` field in interface structure which is used by hash table code.

```
typedef struct ifnet {
    struct ifnet *next;
    if_id id;
    ...
    other fields of interface structure
```

(continues on next page)

(continued from previous page)

```
    ...
} ifnet;


static int if_compare_func(void *_i, const void *_key)
{
    struct ifnet *i = _i;
    const if_id *id = _key;

    if(i->id == *id) return 0;
    else return 1;
}

static unsigned int if_hash_func(void *_i,
    const void *_key, unsigned int range)
{
    struct ifnet *i = _i;
    const if_id *id = _key;

    if(i) return (i->id % range);
    else  return (*id % range);
}

ifhash = hash_init(16, offsetof(ifnet, next),
    &if_compare_func, &if_hash_func);
```

Add object to hash:

```
ifnet *i;

mutex_lock(&ifhash_lock);
hash_insert(ifhash, i);
mutex_unlock(&ifhash_lock);
```

Use hash to find interface by interface id:

```
ifnet *if_id_to_ifnet(if_id id)
{
    ifnet *i;

    mutex_lock(&ifhash_lock);
    i = hash_lookup(ifhash, &id);
    mutex_unlock(&ifhash_lock);

    return i;
}
```

Iterate through hash items:

```
ifnet *if_path_to_ifnet(const char *path)
{
    ifnet *i;
    struct hash_iterator iter;

    mutex_lock(&ifhash_lock);
    hash_open(ifhash, &iter);
    while((i = hash_next(ifhash, &iter)) != NULL) {
```

(continues on next page)

(continued from previous page)

```
        if(!strcmp(path, i->path))
            break;
    }
    hash_close(ifhash, &iter, false);
    mutex_unlock(&ifhash_lock);

    return i;
}
```

hash_hash_str Default function to calculate hash value for strings.

Example of hash function for hashing by string:

```
static unsigned int if_hash_func(void *_i,
    const void *_key, unsigned int range)
{
    struct named_object *i = _i;
    const char *name = _key;

    if(i) return (hash_hash_str( i->name ) % range);
    else  return (hash_hash_str( name ) % range);
}
```

3.2.14 Debugging facilities

Logging

```
#define DEBUG_MSG_PREFIX "vm.exec"
#include <debug_ext.h>
#define debug_level_flow 10
#define debug_level_error 10
#define debug_level_info 0
```

Set up logging subsystem with message prefix to be `vm.exec`, log level for LOG_FLOW messages to be 10 (messages with levels up to 10 are printed, 11 and more are skipped), LOG_ERROR level of 10 and LOG_INFO_ level of 0.

Main kernel logging facility has 6 entry points:

```
LOG_FLOW( level, "format", args )
LOG_ERROR( level, "format", args )
LOG_INFO_( level, "format", args )
LOG_FLOW( level, "format", args )
LOG_ERROR( level, "format", args )
LOG_INFO_( level, "format", args )
```

Additional macro definitions show message in Phantom's debug window and log it.

```
SHOW_FLOW( level, "format", args )
SHOW_ERROR( level, "format", args )
SHOW_INFO_( level, "format", args )
SHOW_FLOW( level, "format", args )
SHOW_ERROR( level, "format", args )
SHOW_INFO_( level, "format", args )
```

If there is a message with no args, add 0 to macros name:

```
SHOW_FLOW0( 1, "Driver started" );
```

CHAPTER 4

Implementation details

4.1 Bytecode Virtual Machine

Phantom bytecode virtual machine (*VM*) is an interpreter which executes Phantom bytecode. Main components of VM are:

- Main code execution loop
- Class loader and cache
- Internal classes engine
- Syscall restart engine
- Root object and services
- Zero state creation code
- Stacks subsystem
- Object allocator
- Garbage collectors

4.1.1 Code execution

Phantom VM executes bytecode in threads. Each thread is native kernel thread and corresponding VM *.internal.thread* object. During snapshot thread state (current *this*, method *ordinal* and *instruction pointer*) is stored in this object, and that is the state thread will be restarted on kernel reboot.

Main code execution loop

Heart of VM. Reads and executes bytecode instrutctions.

Stacks subsystem

There are four stacks in Phantom VM. Object stack, integer (numeric) stack, call stack and exception handlers stack.

Call stack is used when method call or return instruction is executed. Each stack record is a call frame for a running method. Call frame keeps reference to three other stacks for this call frame.

Object stack is main stack for code execution. Most of bytecode operations read input from and put output to this stack. For example, method call reads parameters from object stack and puts func return value here too.

Numerics on object stack

Numeric values can reside on object stack too. Boxed (object form) integers, longs, floats and doubles are used when we store them in objects or pass as arguments.

Numeric stack. This stack can contain 32 or 64 (uses two slots) numeric values: ints, longs, floats and doubles.

Int and float are always 32 bits, long and double are always 64 bits. Longer numerics can be processed in object form only.

Numeric stack contains unboxed numeric binaries and is used for numeric and boolean operations.

4.1.2 Execution environment

Class loader and cache

VM code looks for classes in 3 places:

- List of internal classes compiled into the VM implemenation.
- Boot level class loader which looks for classes in kernel module provided with kernel. It is a usual source for non-internal OS provided classes.
- User level class loader registered on OS startup.

Internal classes engine

Internal classes are classes provided by kernel. All of them have name starting with `.internal`, like `.internal.string`.

These classes connect VM with OS kernel and provide different API entry points.

Most of internal classes have to recreate their objects state on kernel restart.

Syscall restart engine

Blocking or just long running calls from VM code to kernel can not survive snapshot and restart and arranged to be re-run if interrupted by OS reboot.

Root object and services

Phantom object land has one special object called root object.

Root object is used at restart to find some key objects of running OS instance.

- Class objects for very frequently used classes
- List of running VM threads
- Objects that need attention on kernel restart
- User mode class loader
- Root kernel environment (key=val) list
- Kernel persistent stats object
- Class cache map
- Named object directory

Zero state creation code

If OS instance starts for the first time, minimal object land environment has to be created for the first class to be loaded and be able to build the rest of object land.

Such an environment is created *manually* by the VM init code and results in a few very basic classes to be created in persistent memory.

4.1.3 VM and persistent memory

Object allocator

Object allocator is memory allocation code that works in the persistent memory.

Object allocator is working with memory arenas. Each arena is dedicated to either objects of some size or object of some kind. For example, there is arena for integer objects which are all of the same size and are being allocated a lot.

Garbage collectors

There are two garbage collectors in Phantom OS.

Fast and not ideal one is based on reference counts. It is good at releasing short living objects fast and provides for quick object memory turn around.

Slow GC is a classic mark/sweep algorithm, and, depending on persistent memory size, can take very long to complete.

4.2 Phantom Language Compiler

Structure and implementation details of the Phantom OS compiler (*PLC*).

Basis of native Phantom OS personality is a bytecode virtual machine, which, integrated with persistent memory subsystem guarantees Phantom's ability to restart running application code after OS restart.

PLC supports two modes of operation. It can work as a complete compiler (and process source files written in Phantom language), or can be a backend for another compiler.

There are two language support projects currently in progress.

First one is translator for Java class files which is intended to support not only Java language, but also all languages that are able to work on top of JVM, such as Kotlin, Scala, ...

Second one is translator for Python language. It is based on Python parser written in Python and generating intermediate file which is read by PLC and used as input for PLC code generator.

4.2.1 Compiler input

There are three input paths for compiler at the current moment.

- Parser for the Phantom language - this one is production level
- Loader of JVM class files - basic implementation, incomplete
- Connector for Python AST - work in progress

Phantom language parser

Parser processes source code and builds tree (AST) of `Node` objects which represent program in form of (operation (operand) (operand)), where operand is possibly operation too.

Java class file loader

Loads .class files, reads bytecode, rebuilts JVM-style AST, then converts it to Phantom's AST tree, similar to one generated by parser, as described in *Phantom language parser*.

Python frontend connector

Loads special intermediate file which is generated by Python language parser and resembles AST, but has more linear structure.

4.2.2 Compiler pipeline

Process of compilation is divided into the following phases:

- Loading of root classes: .internal.int, internal.string, .internal.class, etc
- Parsing of source file, loading JVM .class, loading other frontend AST
- Loading imported classes
- Preprocessing void status
- Deciding on ordinals
- Finding node types
- Preprocessing nodes
- Generation bytecode
- Generating additional output files (.lst, .lstc, .d, etc)

Code tree prerprocessing

Preprocessing is broken down in a set of phases. Each phase walks down the tree usually processing children before parent is processed.

Preprocessing void status

The goal of this phase is to find out if outer node needs value of inner one. The case when it is true is a code like `a = (b = c)`, where assignment node for `b = c` part still has to return a copy of assigned value for outer `a = (...)` node.

Deciding on ordinals

Current implemenation of compiler assigns method numbers and field positions in objects at compile time. That will change later (OS instance will assign ordinal numbers for all the methods and fields), but now compiler has to do it during preprocessing.

Finding node types

At this phase compiler finds out returning type for each node. Note that this affects code generation in terms of at which stack operations will be executed.

Preprocessing nodes

There is a lot of specific preprocessing in each kind of node, but generally it includes deciding on which stack operation is performed, and whether it needs type conversion, both for numeric types and for objects.

Bytecode generation

This phase is usually quite trivial, because previous phases found out all the relevant information.

4.2.3 Phantom class file

Phantom class file is a simple tagged containers sequence. Each container (file *record*) has type, size and contents. Class file contains just one class.

4.3 Byte Code Reference

This section contains a list of all virtual machine bytecode operations implemented now.

4.3.1 Flow control

jmp

Unconditional jump.

djnz

Decrement top of int stack. If top of int stack is not zero – jump to label.

Note: This operation does **not** pop integer stack!

jz

Jump if top of int stack is zero.

Note: This operation **does** pop integer stack.

switch

`switch shift divisor label...`

```
// calculate expression on int stack
switch 0 1 case1 case2 case3;
// here we'll come on default (fall through)
jmp out;

case1:
// if expression was 0 we'll come here
jmp out;

case2:
// if expression was 1 we'll come here
jmp out;

case3:
// if expression was 2 we'll come here
jmp out;

out:
```

Top of int stack (popped) is diminished by shift (signed), divided by divisor (unsigned) and is used as offset into the jump address table. If result is out of bounds, operator falls through, else jumps to selected address.

ret

Top of object stack is popped and pushed on caller's stack. If stack is empty null object is pushed to caller. All the exception catchers pushed in this call are discarded.

Note that every method returns something, even void. If stack of returning method is empty, VM returns null object for it.

call

`call methodIndex numberOfArgs`

Pops `numberOfArgs` arguments from object stack, than pops object to call method for.

Calls selected method passing given number of args. Top of integer stack of called method will have number of arguments passed. After the return exactly one object (possibly null) will be on the object stack.

static_invoke

`static_invoke methodIndex numberOfArgs`

Pops `numberOfArgs` arguments from object stack, than pops object to call method for, then pops class to call method in. Class is checked to be base class for `this` that was passed.

This instruction is supposed to be used for constructor calls.

dynamic_invoke

```
const «arg val»;
const «arg val»;
const 2; // 2 args
// here must be code to bring this object on stack
const «toString»;
dynamic_invoke;
```

Dynamically invoke method by name.

sys

`sys syscall_number`

Executes this object's embedded method. Object must be of internal class. (See *Internal classes* section)

Nothing can be guaranteed about `sys`. As for now all of them return something, but it is not enforced by interpreter. Though, for normal operation `sys` must return some object or throw an exception.

Actual implementation uses for `sys` the same calling convention as for regular methods.

Sys is not call

It is possible to pass parameters to `sys` on int stack and receive some return there as well, which is differs from `call`, which passes data on object stack only. That is so because `sys` does not create a call frame and thus all the current stack data is available to its code.

Generally, sys is used as the only content of internal class methods.

Current implementation of `sys` is restartable: if `sys` is interrupted (snapshot happens during `sys` execution and OS is rebooted then), it will be run once again. Note that for that to be possible, C code implementing `sys` must be running between `vm_unlock_persistent_memory()` and `vm_lock_persistent_memory()` calls.

4.3.2 Basic stack ops

os dup, is dup

Duplicate stack top – object or integer correspondingly.

os drop, is drop

Delete (pop and throw away) stack top.

os pull

```
os pull displacement
```

Copy object *displacement* steps down from top of object stack on top of it. Pull 0 is equal to dup.

os load

```
os load slot_num
```

Load object reference from this object slot (field), push to top of object stack.

is load

```
is load slot_num
```

Load object reference from this object slot (field), assume object to be numeric, push to top of integer stack.

os save

```
os save slot_num
```

Pop object reference from object stack, store to selected slot (field) of this object.

is save

```
is save slot_num
```

Pop value from integer stack, convert to object, store to selected slot (field) of this object.

os get, is get

```
os get absolute_stack_position is get absolute_stack_position
```

Load object from given position of stack, push to top of stack. Used to access local (stack) variables.

os set, is set

```
set absolute_stack_position
```

Pop object, store to selected position of stack.

os eq

Pop and compare two object references on object stack. If they are the same, push non-zero on integer stack, else push zero.

os neq

Pop and compare two object references on object stack. If they are the same, push zero on integer stack, else push non-zero.

os isnull

Pop object reference. If it is a null, push non-zero on integer stack, else push zero.

4.3.3 Constants

const

```
const “hi there”;
// top of ``object`` stack is string object now
const 32656;
// top of ``integer`` stack has 32656 now
```

Push constant on stack top. Special shortcuts for 0 and 1 exist. Strings are Unicode UTF-8.

const_pool

```
const_pool 22;
```

Load constant number 22 from constants pool. Constants pool is loaded from class file. Practically this operation is used instead of string constant, but supposed to be used for other purposes too.

4.3.4 Getting special objects

summon this

Puts `this` object reference on stack.

summon thread

Puts current thread object reference on stack.

summon null

Put null object on stack.

summon class_name

```
summon “internal.string”;
```

Push corresponding class object on stack.

4.3.5 Integer stack operations

Integer stack supports long/float/double operations too. By default operations are performed for integer type. For other types byte code operation must be prepended with a prefix.

prefix_long, prefix_float, prefix_double

Select numeric data type for the next integer stack operation.

```
opcode_prefix_double;
i2o;
// *object* stack now has ``.internal.double`` object
```

These prefixes change default integer type for integer stack operations to corresponding type. Next integer stack operation is performed for given type. If type is 64-bit one (long or double), 2 stack slots are used for each operand or result. Note though, that all compare operations return int no matter which prefix you use.

fromi, froml, fromf, fromd

```
const 10;
prefix_float;
fromi;
// numeric stack now has float value 10.0 on top
```

Convert int stack value from given type to current (int by default, or as set by prefix).

i2o, o2i

```
const 10;
i2o;
// ``object`` stack now has integer object
o2i;
// value of integer object moved back to numeric stack
```

These operations can be used to move numeric data between integer and object stacks.

isum, imul

```
const 10;
const 5;
isum;
// int stack has 15 on top
const 10;
imul;
// int stack has 150 on top
```

Addition and multiplication.

isubul, isublu, idivul, idivlu

Integer subtraction and division. Operand order:

- *ul* Subtract Upper from Lower.
- *lu* Subtract Lower from Upper.

iremul, iremlu

Division remainder (modulo).

ior, iand, ixor, inot

Bit wise operations on integer stack top.

logical operations on integer stack top.

ige, ile, igt, ilt

Comparison operations. Prefix sets just operand types, result is always integer.

ishl

Shift left.

ishr

Shift right signed. Fills with sign bit.

ushr

Shift right unsigned. Fills with 0.

4.3.6 Exceptions

Phantom virtual machine supports quite classic set of exception handling operations. Exception throw instruction accepts any object to be thrown. Catcher is waiting for object of given class or any of its subclasses.

push catcher

```
summon "internal.string";
push catcher string_thrown;
// code
ret;

string_thrown:
// on exception of string type we'll get here with thrown object on stack top
// do cleanup
ret;
```

Top of stack must contain class object. Exceptions of that class will be catched here and control will be passed to a label if such exception is thrown. Thrown object will be on stack top in this case. In general no other assumptions about stack state can be done.

pop catcher

Last pushed catcher will be deleted. Note that all method's catchers will be automatically popped on return.

throw

```
const "we have a problem here";
throw;
```

Object on top of stack is thrown (exception is raised). If stack is empty - will throw special system-wide object (null?), if already on top of call stack - will kill current thread in a bad way.

4.3.7 Special operations

nop

Of course.

debug

Prints string argument (if any), and top values of integer and object stack. Integer argument (mode) is one byte and upper bit is used internally, so don't pass anything > 0x7F or < 0 – will be stripped.

- Bit 0 (& 0x01) of mode byte turns on debug instruction printing.
- Bit 1 (& 0x02) turns it off.

Not really used now.

new

```
summon ".internal.string";
new;
```

Create a new object of given (stack top) class. String, in this case. No constructor called automatically. Compiler must generate code to call constructor.

cast

```
// push object to cast
// push target type (class reference)
cast
// object reference now refers to specified class's interface
```

Pops target type from stack, pops object, checks that it has type is among parents, pushes object back.

When interfaces will be implemented, will do more - add or remove proxy?

stack_reserve

Reserve space on object and integer stacks. Used to set stack place for stack variables.

arg_count

Parameter of this op is number of args this func waits for. Operation compares it with integer on istack, and throws message if values differ.

lock_this, unlock_this

Not implemented. Supposed to lock/release mutex attached to `this` as satellite.

general_lock, general_unlock

Not implemented. Supposed to lock/release mutex at stack top.

CHAPTER 5

Concepts and Problems

In this chapter the most discussed problems of Phantom OS are discussed.

5.1 Conceptual difficulties of persisten OS

5.1.1 Program update

How can one update program code if program never stops?

It seems that it is impossible to answer this quiestion. But we will try.

First of all, it must be noted, that code in Phantom, which is allways part of class, can be versioned. Each class can exist in multiple versions in one OS instance. By default when variables are instantiated, newest version of class is used. But older ones still exist as long as exist old objects that refer to such a class.

It lets us to have simple way to do the first step - bring new code to system. Developer can just install a new version of code and it will work smoothly - new objects will be created with new implementation, old ones will continue to use code they was created with.

But still - we want already running programs to use new code, do we? There are many possible ways to do so.

1. Actually ask user to... stop program and restart it. The fact that Phantom does not stop programs does not prevent us from doing it. We can arrange code in a classic ways - here are data objects, and here is a program that uses that data objects. Restarting such a program is re-creating objects it is made of.

   It looks that we loose in this case all the better features of persistent OS, but it is not so. As long as program runs it still can keep its state in persistent objects and gain from doing so.

2. Arrange code so that there is separate state objects and set of processing classes. If we will be creating instances of processing classes on each use, update of code will go seamlessly for us.

   ```
   rgbImage = new ImageProcessor( grgbImageObject ).convert()
   ```

   Such calls can be hidden by public interface of serving class.

3. Provide means for conversion. Each new class can have copy constructor for creating object of that class from the object of previous class version. OS can, in this case, run over an object tree for your application and perform object replacement, calling your copy constructor for each of objects bottom to top.

4. Convert in place without any support from code.

   The biggest problem of code replacement in run time is references to fields. If object structure changes, field displacements in objects can change. If we replcae code leaving data intact, new code will access wrong fields.

   Having field names and displacements for both old and new code OS can update objects, moving data in new positions and clearing fields which are new to this version of code.

   This method seems to be the best one. The only thing that it requests from application programmer is to be tolerant to absent field values.

   Or, again, provite an update constructor.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

O

P

R

S

T

U

V

W