

# Phantom OS

Fosdem 2020



# Why new OS

- ✦ The only OS concept that exists is Unix. Even Win NT is Unix like, more or less.
- ✦ Unix is based on what was possible half a century ago
- ✦ Unix is based on what was possible, not what is needed
- ✦ Traditional big kernel is not a good solution, hence all attempts to create a microkernel OS or, at least, move part of OS services out of kernel



# Why not microkernel

- Microkernel relies on object based cross-address space IPC
- All attempts to build remote object access failed because of transaction cost
- The question is - do we really have to have separate address spaces? Why?
- As Java/.Net/name your modern language application servers show, if we can control address calculations, we can protect information on per object basis in one big address space.



# Why should program die?

- There's no reason.
- If we replace microkernel environment with global address space containing managed code, components can communicate at **no cost**, by simply sharing pointers one with other.
- But then we have a problem if we keep pointer to some service, and that service is not yet restarted after OS reboot.
- Well, lets design OS so that **applications do not have to stop** when OS kernel reboots.



# What Phantom OS is (1)

- ✦ Persistent virtual memory, one global address space. Memory state is restored on OS restart.
- ✦ Bytecode virtual machine (more or less language agnostic), running in persistent memory.
- ✦ **Very fast reboot**: OS does not have to rebuild environment from scratch, running code is just paged in from disk. Paging is cheap.



# What Phantom OS is (2)

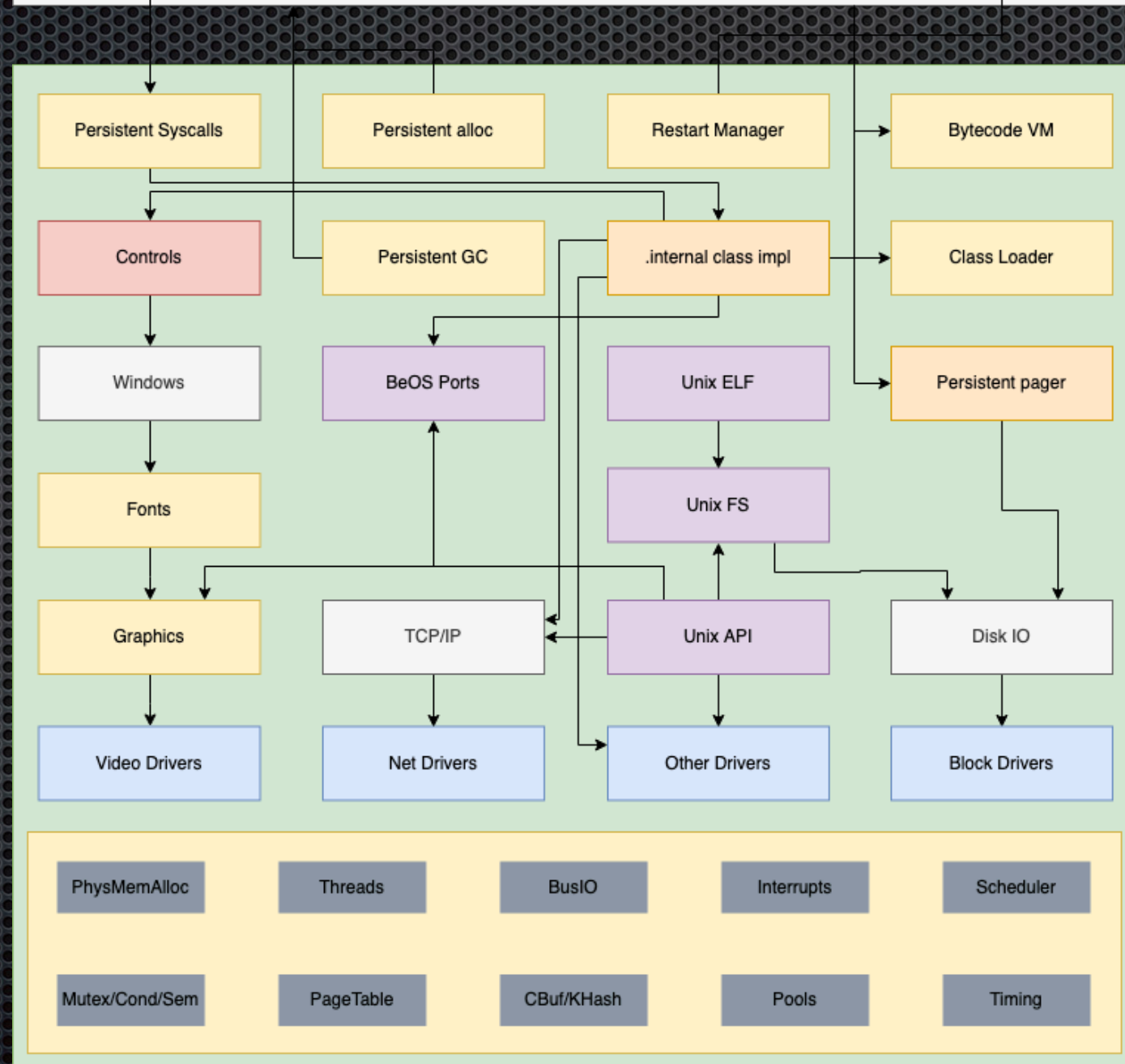
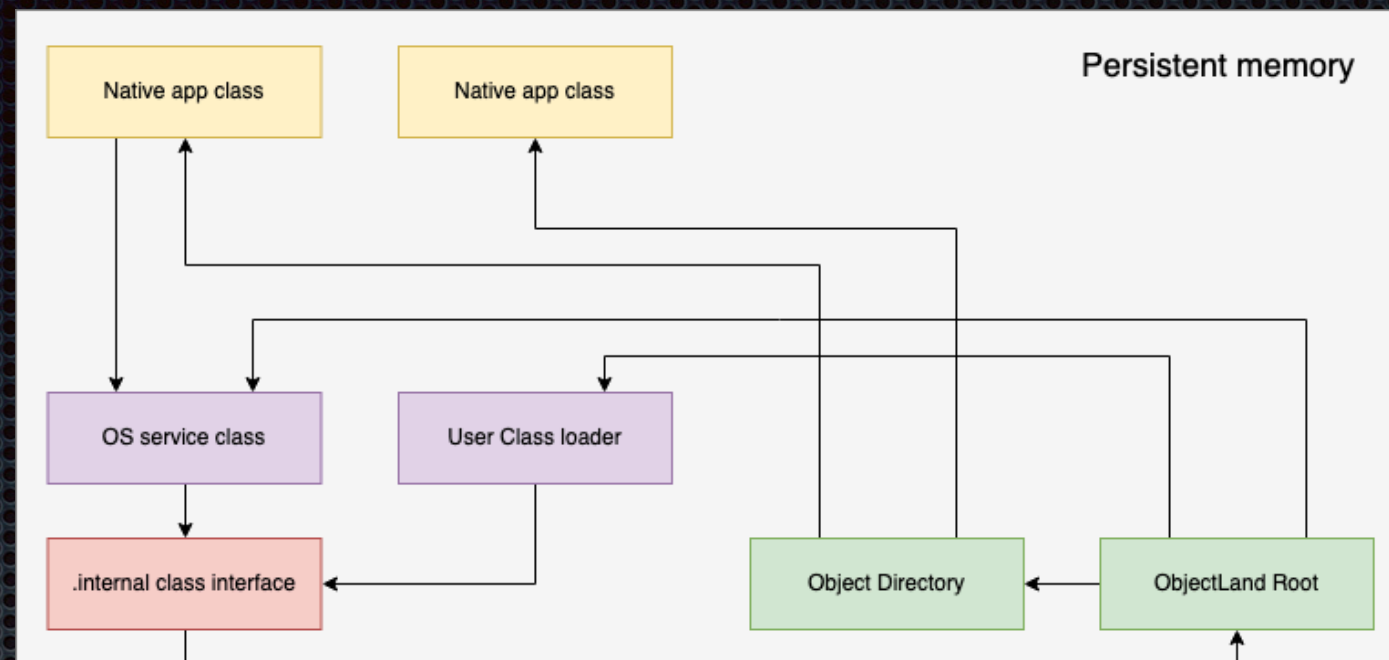
- ✦ **Fault tolerant:** OS state guaranteed to survive blackouts and hardware malfunctions. If machine is broken, OS image can restart from last checkpoint on a new hardware. It is possible to keep OS state on remote server.
- ✦ **Secure:** virtual machine is designed to protect caller from callee completely, method can't access caller's stack. Actually code can access nothing but this and arguments.



# Not POSIX, but...

- Native Phantom OS personality is **not POSIX**. It is seen to native code as an object-oriented library partially implemented in kernel.
- Native Phantom code can be written without access to file system. As **all variables are persistent**, there's no need to save state to file to survive OS restart.
- Still, there is FS support in Phantom.
- There's even **simple and limited POSIX subsystem**. Not persistent. Yet?







# Save state

- ✦ If we want user mode code to survive OS reboot, we have to **save complete user land state**.
- ✦ Make sure that **state is in memory** (not in registers, kernel memory, etc)
- ✦ Dump all virtual memory
- ✦ Not stopping world. Frequently. Efficiently.



# Snapshots

- ✦ Phantom OS persistence is based on snapshots. Each snapshot is a **synchronous copy** of virtual memory state.
- ✦ Taking snapshot does not stop or pause OS or user code threads.
- ✦ There are two (or more) snapshots exist on disk so that fault during taking a snapshot does not break following OS restart. Snapshots reuse unchanged data from previous snapshots.



# Internal classes

- ✦ Userland runtime environment is object oriented. Anything up to single **integer is an object** of some class.
- ✦ Some classes are plain (user) ones, some are **internal (native)** and **implemented as kernel code**.
- ✦ Internal classes can request to be called on OS kernel restart before all the user land threads will continue. It can be used to **restore corresponding kernel state** for such object to continue its service.



# Special cases

- **Extra fast restart:** it is possible to keep snapshots in RAM, MRAM, or interleaved flash. This way restart can be possibly done in extremely short time.
- **Low power consumption:** Snapshot generation period can be controlled and if extended, disk subsystem power consumption can be reduced. Additionally, ping style IO does not consume CPU at all. No FS structure to update, no IO buffers to move.
- **Extra reliable:** It is easy to keep a completely synchronised remote copy of snapshots to be able to restart even if hardware is destroyed completely.



# Kernel is non-persistent

- ✦ Kernel restarts from scratch, its state is not saved.
- ✦ It means that no blocking syscalls possible: it will be impossible to recreate state of such call.
- ✦ As a workaround, blocking calls interrupted by OS shutdown will be restarted on next OS start.



# Not so simple

- ✦ Userland is **huge data segment** filled with objects belonging to different threads, users or even other network nodes.
- ✦ No manual memory freeing can be used. Your object can be **shared** with other thread or over the net. Just GC.
- ✦ Size of data segment is **nearly size of disk** - all the data is memory mapped for native code. Think of terabytes.
- ✦ Doing a **GC** for such a huge memory is unusual.



# 2 GC

- **First GC** has to be **fast**, simple, and be able to release about 90% of garbage, especially - short living objects. We user refcount.
- **Second GC** can be (extremely) **slow**, but has to be able to release any kind of garbage (loops).
- Second GC works on a... snapshot. This way it can be implemented in a stop the world fashion. Its world is already stopped.
- Those two must never touch same part of memory.



# Blocking sys calls

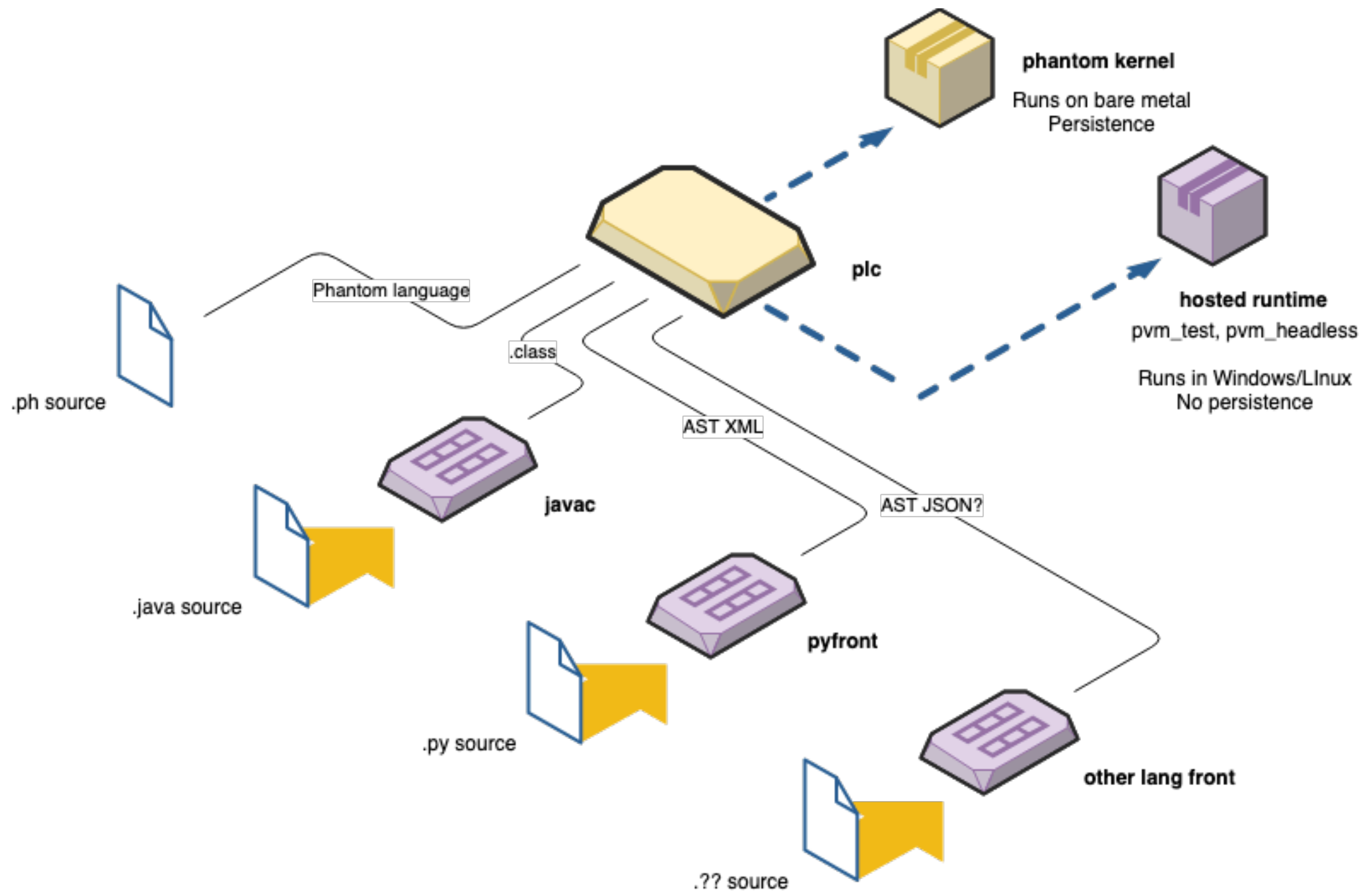
- Impossible without special support
- **Kernel is not persistent**, if thread is in kernel code, its state can't be restored
- Modifications to the persistent memory from kernel possibly non-atomic
- Blocked system call **restart** as a solution
- Blocked sys call is **atomic** from the snapshot system point of view, either completed or not started. Parameters are saved until it is completely done.



# Migration path

- ✦ Work in progress - convertor of **JVM byte code** to Phantom OS byte code. Current state - it is possible to convert simplest code. Main issue - basic class lib classes for Phantom.
- ✦ Work in progress - direct compilation of **Python** to Phantom.
- ✦ Experiment: **Managed C** code execution. Just started.







# Portability

- ✦ Most mature code is for x86 only.
- ✦ **Arm** port: was done 4 years ago, kernel regression tests passed, stopped on writing device drivers. Not updated, needs update for current CPU modifications.
- ✦ **MIPS** port: started, code compiled but not all kernel regress tests passed. Not active.
- ✦ General **64/128 bit** support: compiled for 64 and 128 bits architectures to find out possibly problematic places.



# More

- ✦ Realtime scheduler, some of kernel modules are realtime threads.
- ✦ Network paging, paging IO UDP based server for Linux
- ✦ SMP ready, but not tested regular basis, not stable



# CI and regression tests

- CI includes **regression tests** for kernel primitives in kernel mode, kernel primitives in user mode.
- Manually run compiler regression tests, byte code virtual machine regression tests.
- User mode environment used to test user mode code.
- Kernel is tested in QEMU, some versions passed 100+ abrupt reboots restoring persistent memory state. Simple applications continued to run.
- Kernel is able to boot on **real hardware**. Manual tests.



# Code update

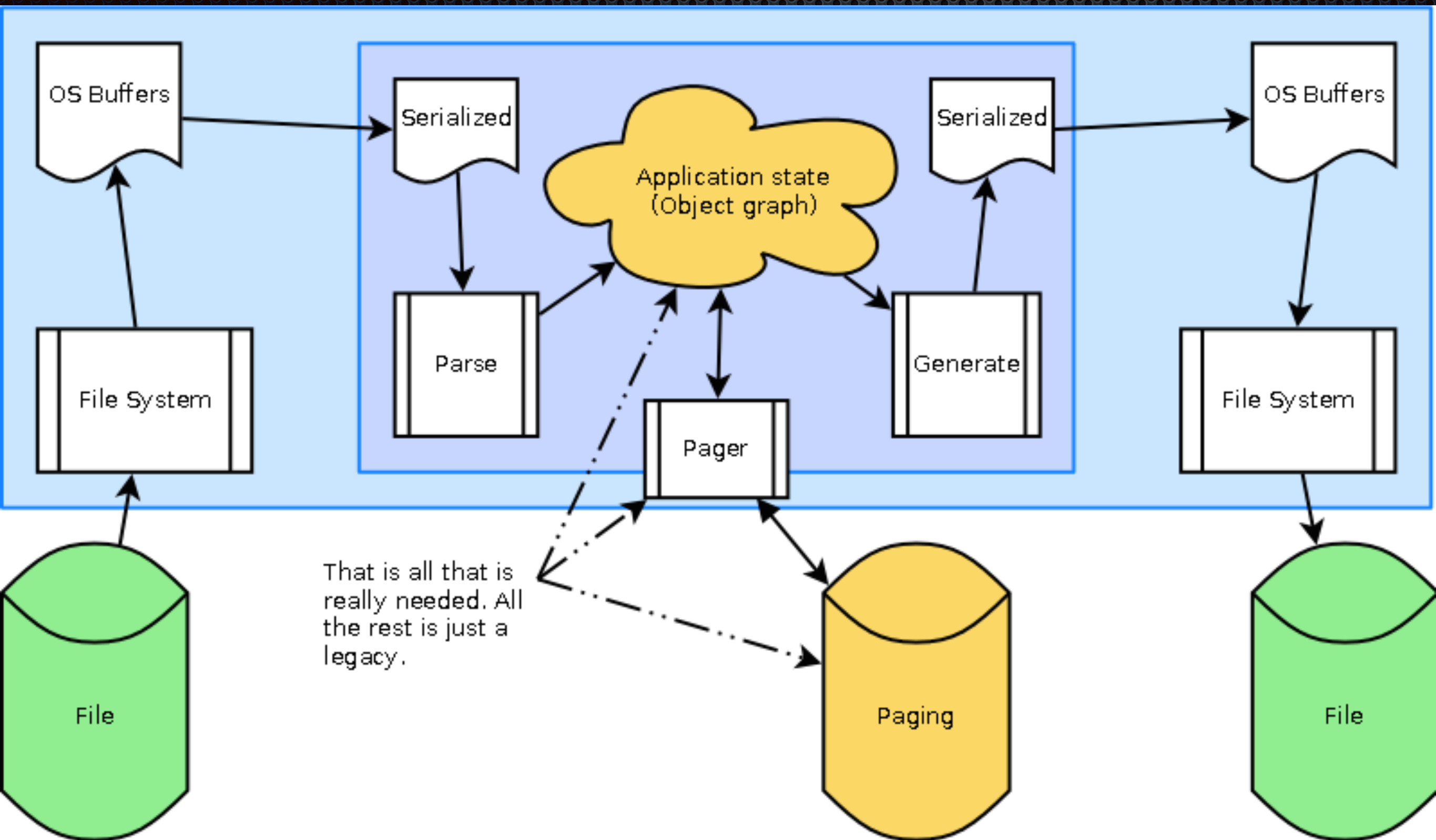
- ✦ In persistent environment **updating** application code is not a simple task.
- ✦ **Class versions** - simple but helps just in some cases.
- ✦ **Hot patch** - not yet implemented, not simple.
- ✦ Code organisation matters. Separate content from processing classes.



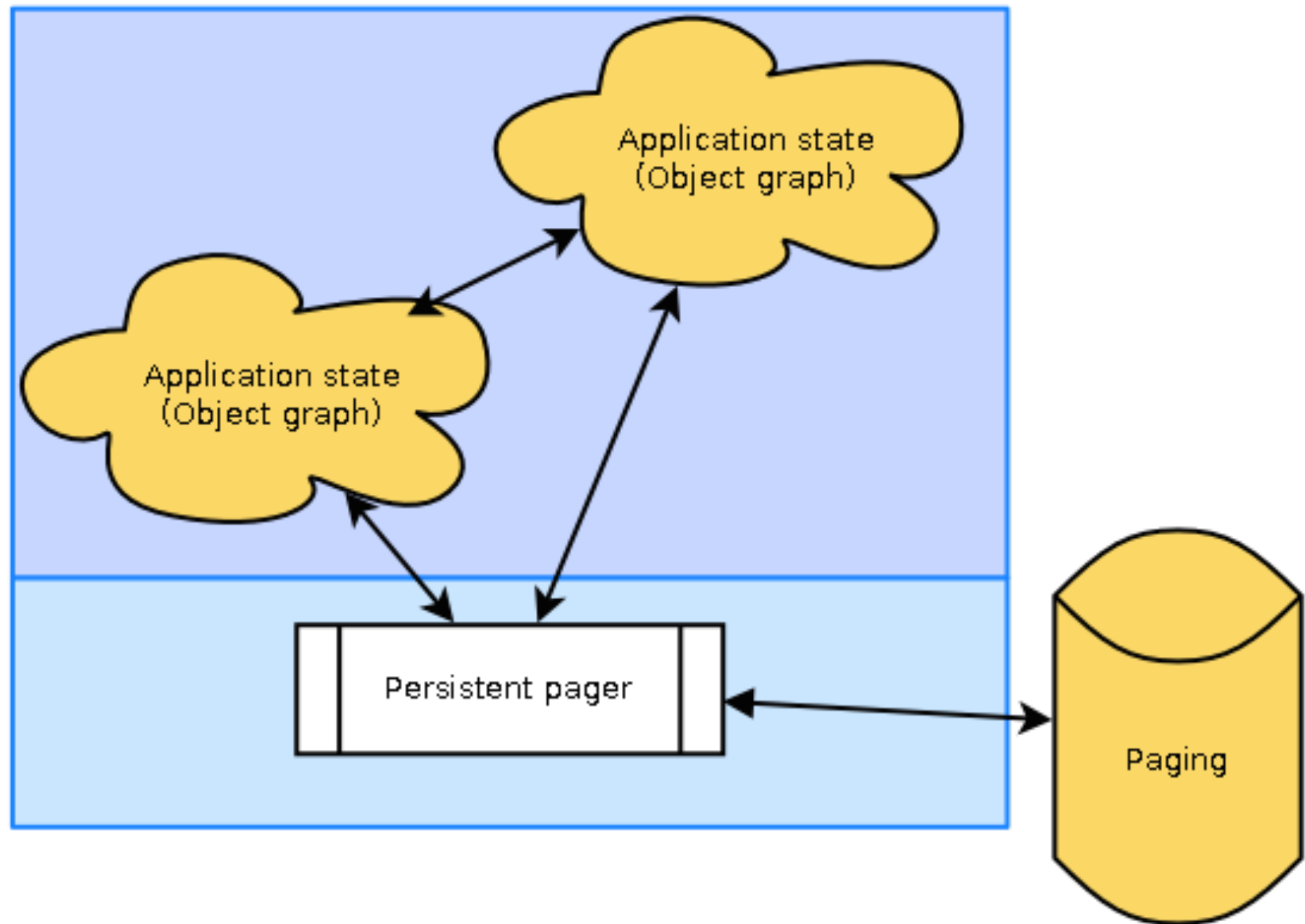
# What do I look for

- ✦ Collaborants and contributors, kernel, VM, compiler, UI, drivers, CI and tests, etc.
- ✦ Projects to use Phantom OS as base or part
- ✦ Partnership on project or parts. For example, byte code virtual machine can be used apart.





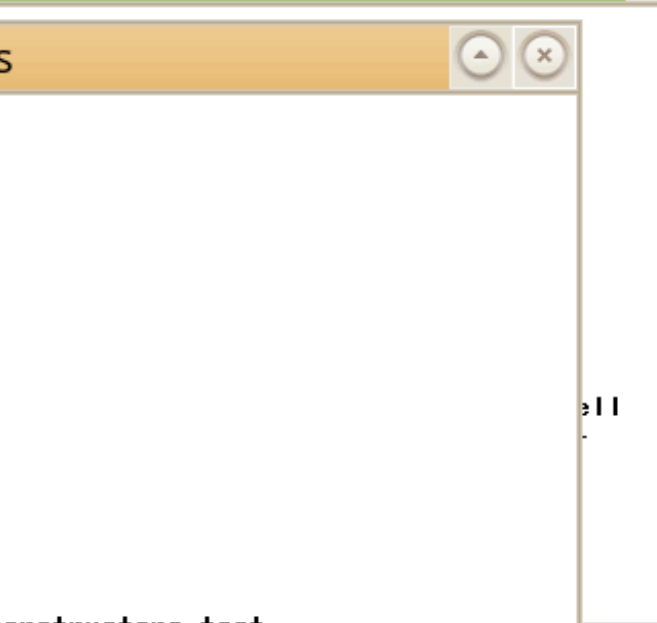






```
stopped
slog send '<0>0ct 16 21:42:48 10.0.2.15 snap:
ill, say 'cheese!' ...
slog send '<0>0ct 16 21:42:48 10.0.2.15 snap:
ou ladies
slog send '<0>0ct 16 21:42:48 10.0.2.15 snap:
yslog send '<0>0ct 16 21:42:48 10.0.2.15 snap:
inalize_snap
```

The screenshot shows the 'Physical memory' window in WinDbg. The title bar reads 'Physical memory'. The menu bar includes 'View: threads stats windows profile' and '? - help'. The status bar displays 'Step 6059, uptime 0d, 00:13:03, 0 events' and 'Time 2019/10/16 21:42:59 GMT, CPU 0 56% idle'. The main content area is mostly black, with a horizontal band of multi-colored horizontal lines (red, green, blue, yellow) in the lower half, indicating a memory dump or crash data.



The screenshot shows a window titled "VM TTY Window" with a green title bar. Inside the window, there is a terminal-like interface with an orange header bar that says "tests". The terminal content shows the execution of a test suite:

```
all
.
suit
chy constructors test
erarchy constructors test
```

A mouse cursor is visible on the right side of the window.

# VM Shell

```
Weather win: curl = 9
Weather win: sleep
Weather win: curl
Weather win: curl = 9
Weather win: sleep
Weather win: curl
Weather win: curl = 9
Weather win: sleep
Weather win: curl
Weather win: curl = 9
Weather win: sleep
Weather win: curl
Weather win: curl = 9
Weather win: sleep
Weather win: curl
```



# Usual Q & A

- ✦ Q: Now you can't restart failing application?
- ✦ A: You still can, but application can be sure that it is not stopped by kernel reboot. Just your will.
- ✦ Q: Hardware based nonvolatile RAM makes any OS persistent.
- ✦ A: Actually, it is not so simple. Hardware state is not saved in RAM anyway, you have to handle restarts even is RAM state is intact.



# Contacts

- ✦ Dmitry Zavalishin, [dz@dz.ru](mailto:dz@dz.ru)
- ✦ <https://github.com/dzavalishin/phantomuserland>
- ✦ <https://phantomdox.readthedocs.io>

