ReactOS (no) es Windows

Windows internals and why ReactOS couldn’t just use a Linux kernel

ReactOS is (not) Windows

- ReactOS *is* Windows
  - Runs Windows applications
  - Runs Windows drivers
  - Looks like Windows
- ReactOS *is not* Windows
  - ReactOS is a free, open source project
  - ReactOS reuses open source code from other projects
  - *You* can make “your own Windows”

Who am I?

- **Michele “KJK::Hyperion” C.** from Italy
  - Last Italian team member
  - Senior ReactOS developer (since 2001)
  - hackbunny@reactos.org

What does Windows mean

WINDOWS ARCHITECTURE

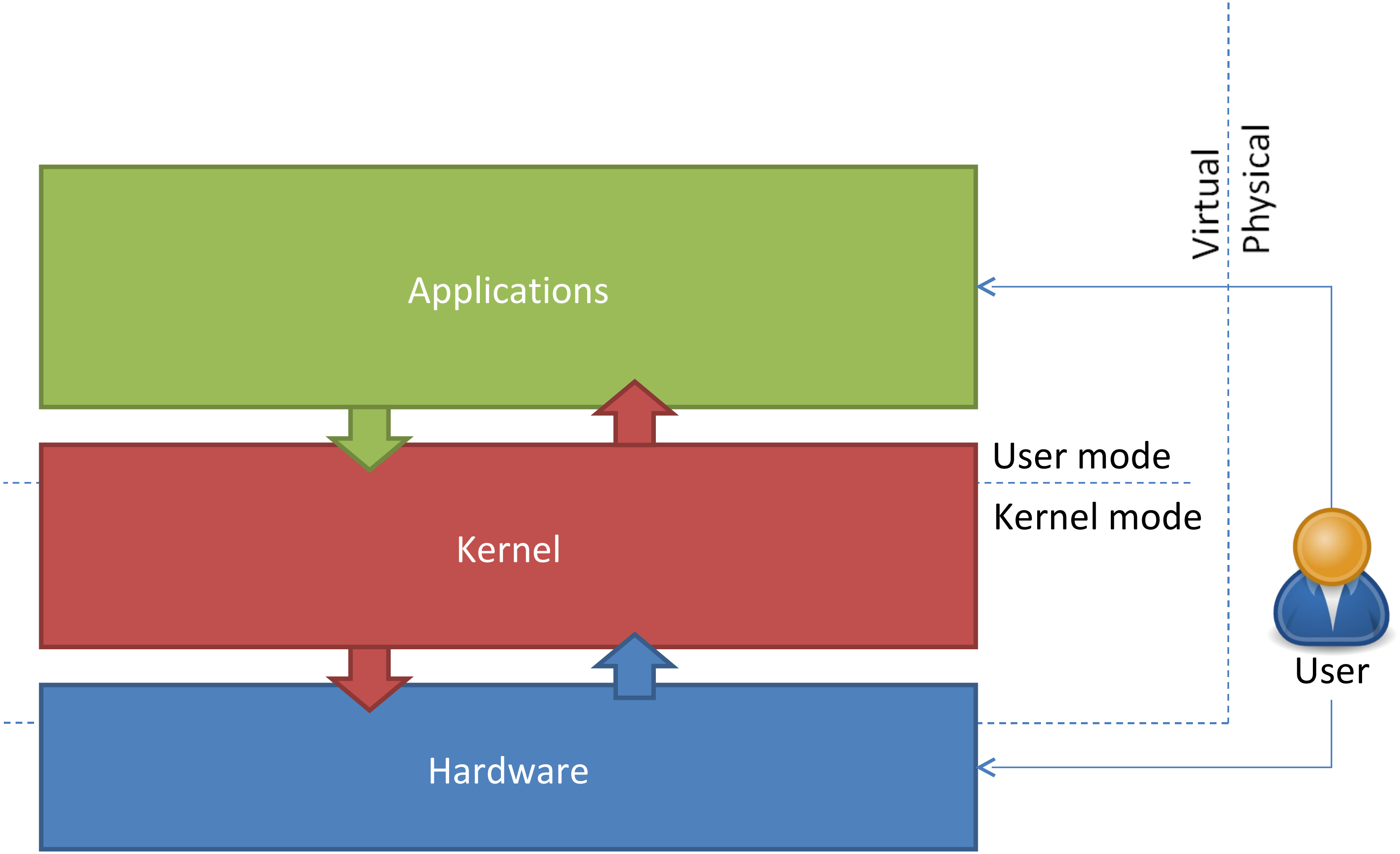
ReactOS *is* Windows

- ReactOS has the same identical architecture as Windows, for **maximum compatibility**
  - Windows drivers require a Windows kernel
  - Many applications (firewalls, antivirus, media players, PDA sync software, etc.) come with special drivers
- Windows architecture is quite different from Linux and not as well known
- Let’s start from the basics…

Operating system architecture

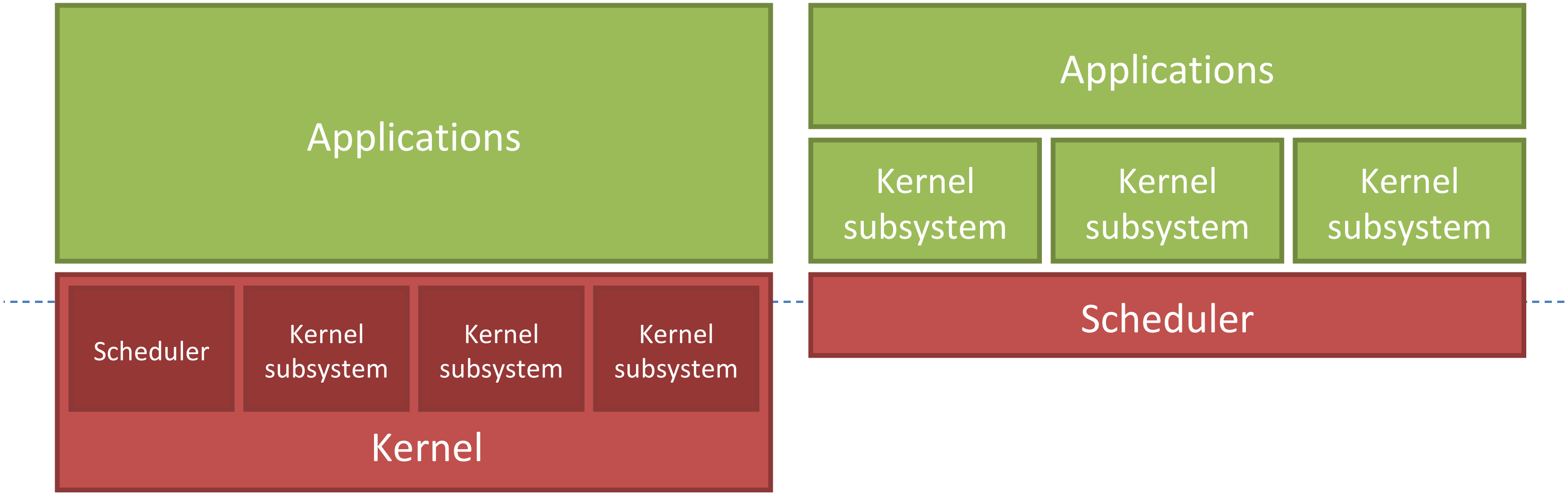
- Abstraction of **CPU time and context** (processes, threads, signals, etc.)
- Abstraction of **memory** (virtual memory, paging, stacks, heaps, etc.)
- **Separation between system and applications** through CPU’s built-in memory protection (user mode vs kernel mode)
- **Separation between hardware and applications** through CPU’s built-in I/O privilege mechanisms
- Mechanisms to **bypass OS protection features** in a controlled, secure way (system calls, security subsystem, etc.)

Operating system architecture



A note on microkernels

- A **monolithic kernel** abstracts all hardware resources. All subsystem run together in supervisor mode. **Almost all operating systems are monolithic**
- A **microkernel** only abstracts CPU time (scheduling). Kernel subsystems are regular applications running in user mode, for more stability at the cost of performance. **Very few operating systems are pure microkernels** (too impractical, almost all CPU architectures are designed for monolithic kernels, etc.). **Windows is *not* a microkernel** (more on that later)



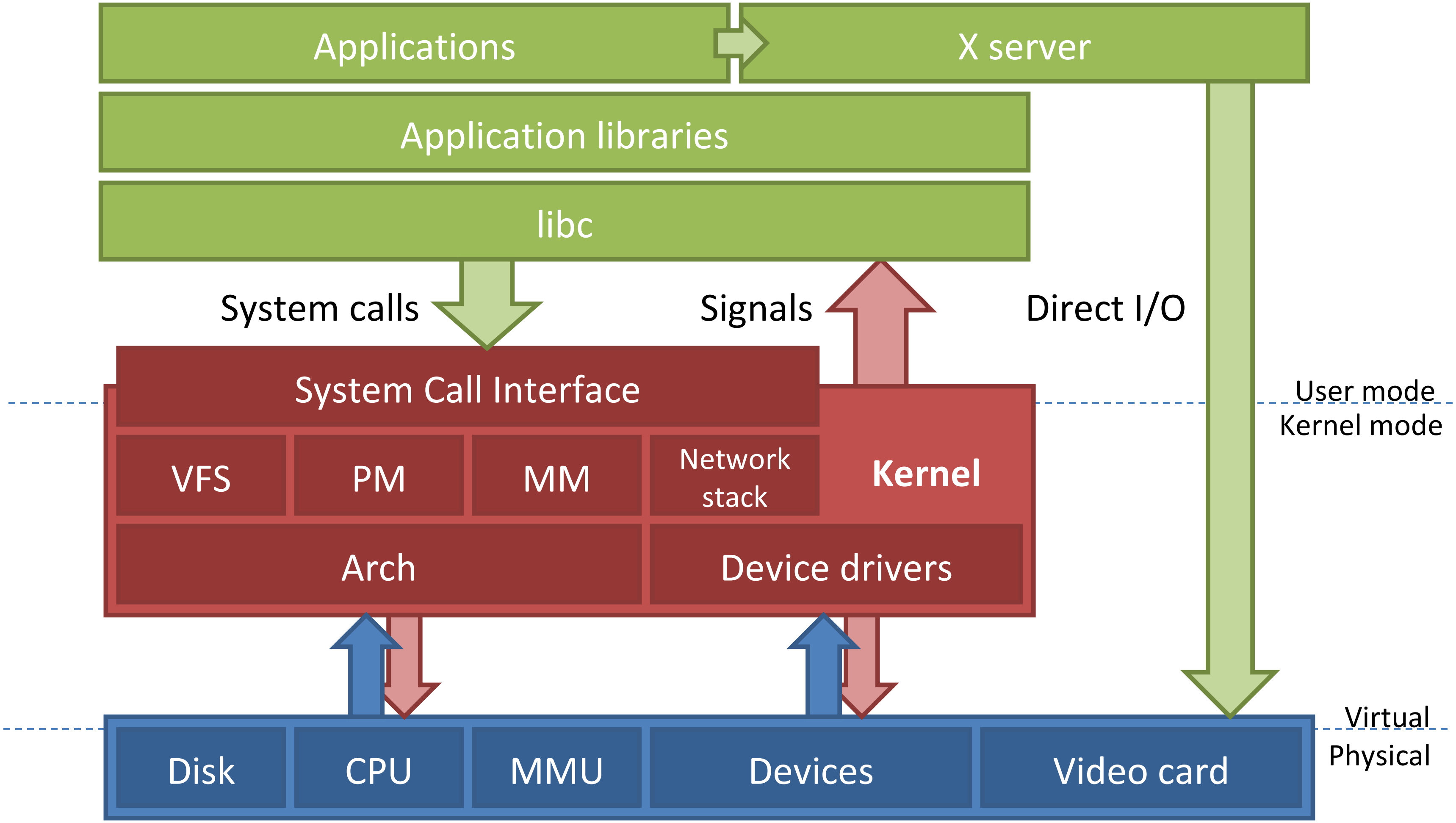
Regular (monolithic) architecture

Microkernel architecture

Linux architecture

- **Monolithic kernel**. No kernel ABI
- **UNIX process management and security model**
- **Native networking support**
  - Sockets, pipes
  - select, poll, etc.
- **Filesystem abstraction** (VFS)
- **UNIX API** (libc) on top of a small UNIX-like system call and signals interface
- Other APIs (audio, application setup, desktop environment integration, cryptography, etc.) are ***de facto* standards from third parties**
  - The graphic subsystem (X server) is in a category of its own. The kernel has “backdoors” to let the X server talk directly to the hardware, to keep the complexity of video drivers outside of the sensitive environment of kernel mode

Linux architecture



Windows (NT) architecture

- **Monolithic kernel**. Relatively stable kernel ABI
- Kernel design is almost identical to DEC RSX-11 and **VMS**, with DOS, OS/2 and Windows 95 influences
  - RSX-11, VMS and Windows NT were designed by the same engineer (Dave Cutler)
  - Windows NT was initially developed as a new kernel for OS/2
- No device abstraction in the kernel itself. Abstraction is provided by standard system drivers (*class* or *port* drivers)
- **No network support in the kernel itself**. select/poll is not a system call, but an ioctl to the "socket filesystem"
  - **Sockets and pipes are provided by two special filesystems**
  - Further user-mode layer of abstraction sockets: Winsock used to be a third-party component (e.g. Trumpet Winsock)
- **Native graphics and windowing subsystems (running in kernel mode)** with a standard API
- **Rich, high-level APIs** of all sorts (cryptography, desktop environment, etc.)

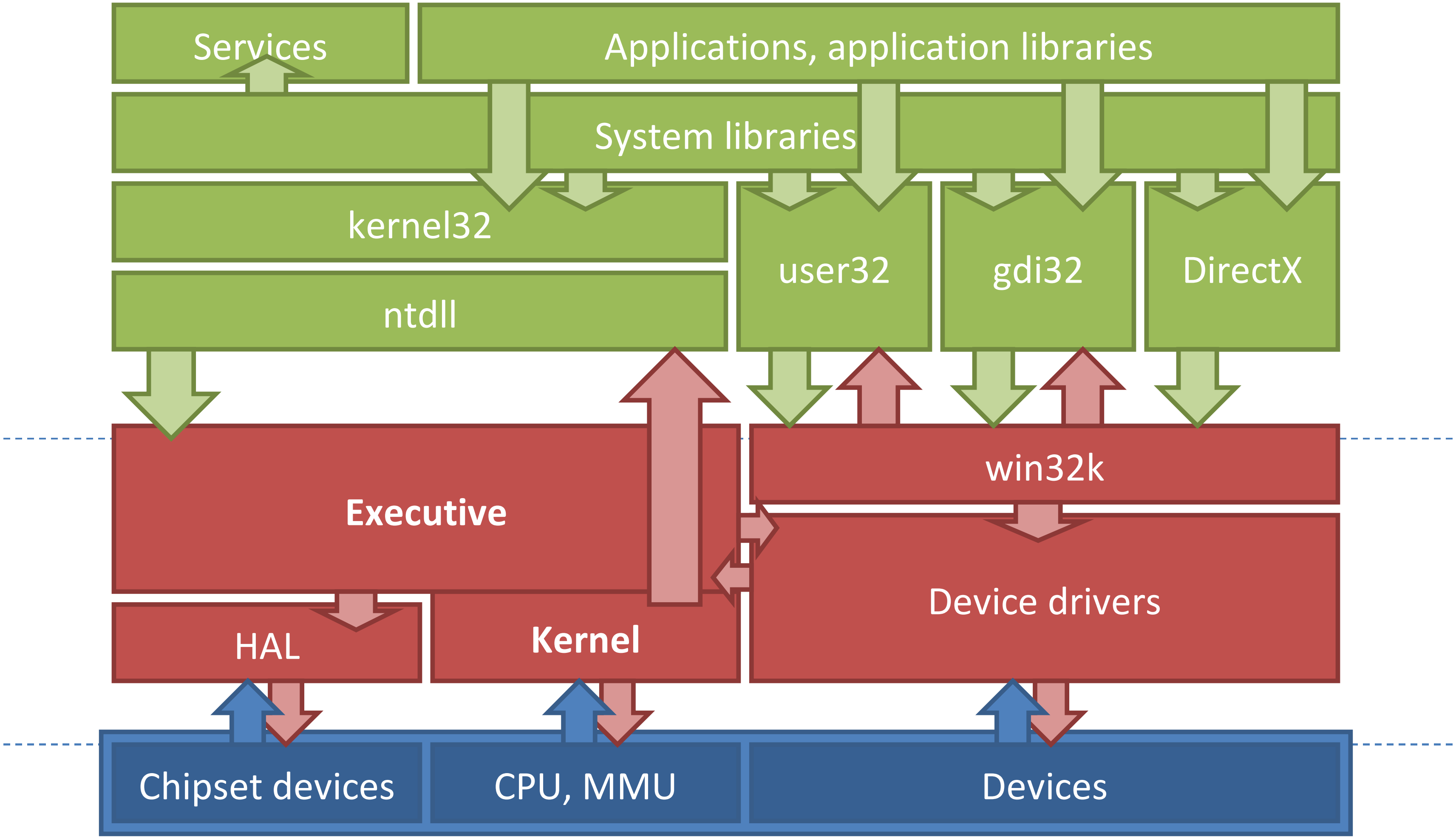
Is Windows (NT) a microkernel?

- Was initially designed as a microkernel. Deemed impractical; turned into a monolithic kernel long before release
- Microkernel legacy in Windows:
  - Kernel is still logically split in two:
    - **Kernel**, which implements the scheduler
    - **Executive**, which implements everything else (process management, I/O, security, etc.)
    - Executive calls Kernel, but never the other way around
  - Until Windows NT 4, the graphics and user interface subsystems (USER and GDI) ran in user mode
  - Many system APIs are provided by user-mode services

Unique Windows architecture features

- Chipset devices (timer, interrupt controller, power management, buses, firmware, etc.) are abstracted by a kernel component called **Hardware Abstraction Layer** (HAL)
  - ACPI vs non-ACPI is just a different HAL
  - The ReactOS port to the XBox was a regular x86 ReactOS with an XBox-specific HAL
- No signals; **standard exception model** instead (“SEH”, shared with VMS, OS/2 and Tru64)
- **Reverse system calls** (callbacks): windowing and graphics subsystem can call back into user mode
  - The user-mode and the kernel-mode parts of the subsystem used to run in shared memory in their original implementation (Windows 95)
  - Too unsafe for Windows NT; emulates a secure but compatible shared memory environment with some “tricks” (like callbacks)

Windows architecture



ReactOS is (not) Wine

WINE AND REACTOS

ReactOS is (not) Wine

- “If ReactOS is just a kernel for Wine…
  - … what do we need it for?”
  - … why isn’t it finished yet?”
- As always, things are more complicated than they appear…

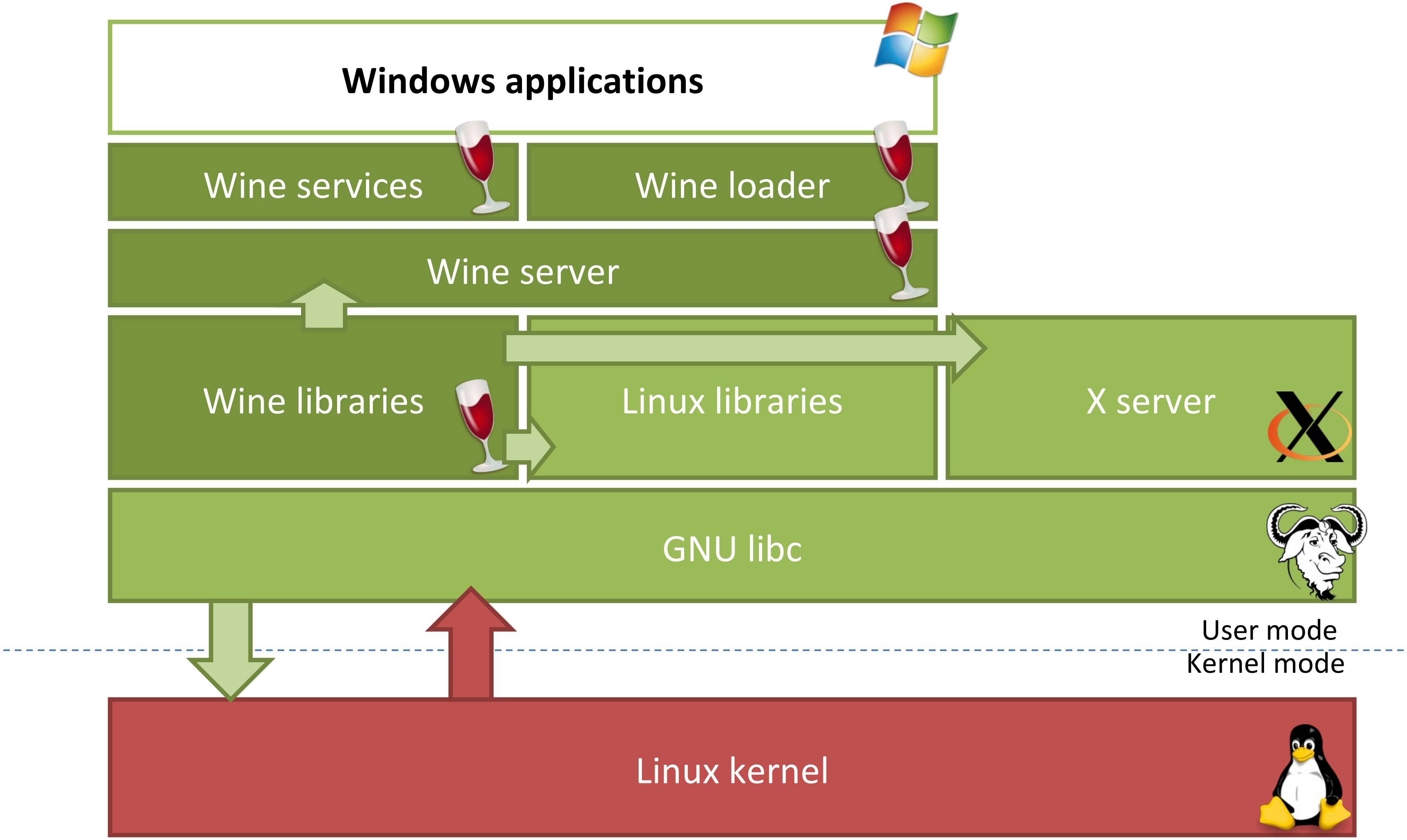
Wine and ReactOS

- ReactOS has a lot in common with Wine, and we can share a lot of code with them…
  - Enough common goals:
    - Installing Windows applications
    - Running Windows applications
- … but…
  - Too many different goals
    - Running on Linux vs running on hardware
    - Whether to support Windows drivers

Wine on Linux

- Windows applications can only be loaded by a Wine utility (the Wine loader)
- Windows applications and DLLs are dynamically linked to Wine reimplementations of Windows system DLLs
  - Most Wine DLLs are regular Windows DLLs compiled as Linux code
  - Some are internally Linux libraries, depending on other Linux libraries. Linux libraries are transparent to Windows applications – they act as system calls in all respects
- A service process (Wine Server) replaces the Windows kernel for the management of shared resources

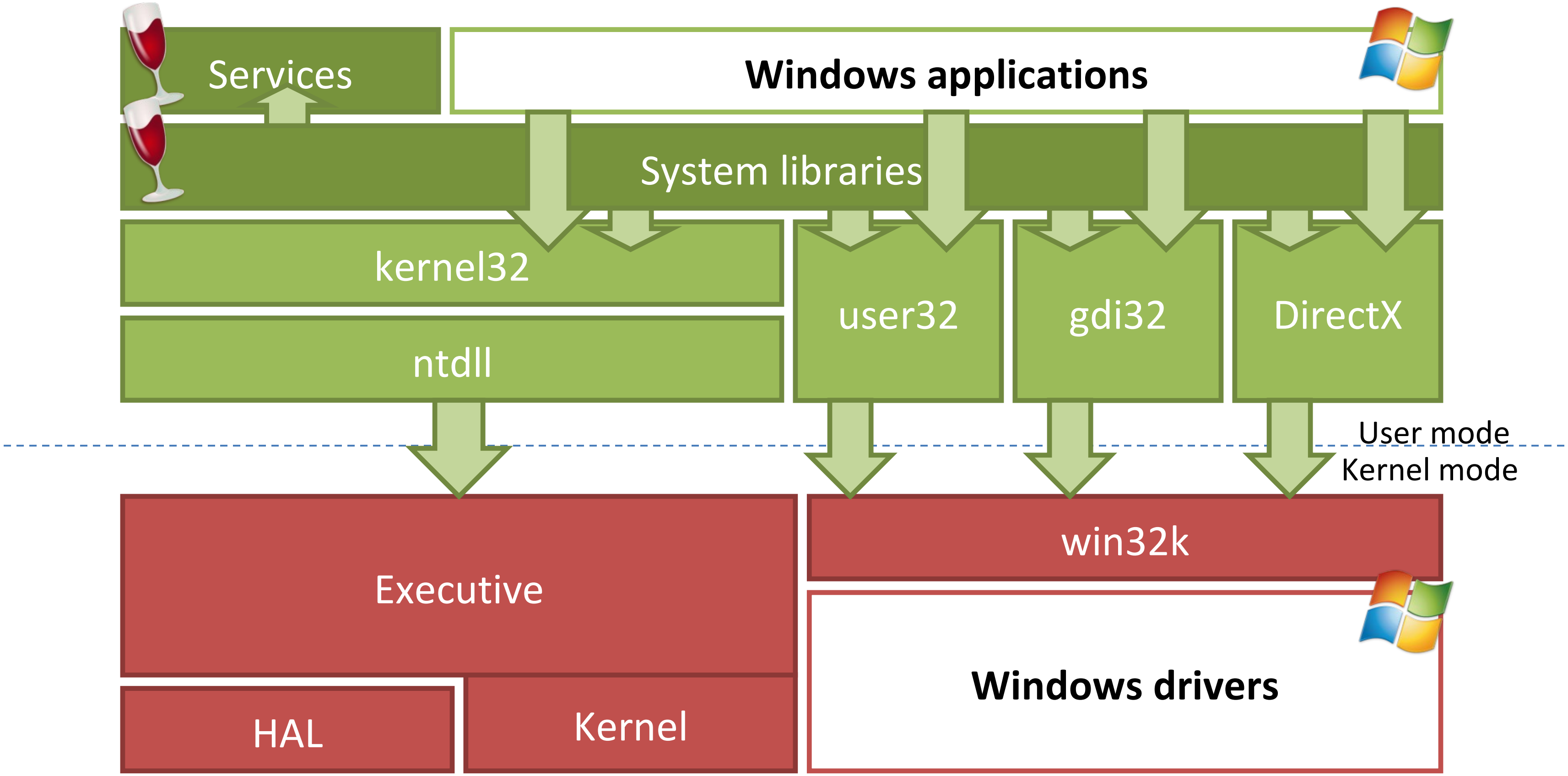
Wine on Linux



Wine on ~~Windows~~ ReactOS

- Windows applications are loaded *directly* by the ~~Windows~~ ReactOS kernel
- Windows applications and DLLs are dynamically linked to Wine and ReactOS reimplementations of Windows system DLLs
  - We can only use Wine DLLs that don't depend on Linux libraries
  - This includes important libraries like user32 & gdi32 (windowing and graphics APIs, depending on X server on Linux), wininet (HTTP and FTP client, depending on OpenSSL on Linux for HTTPS), etc.
- **ReactOS reimplements a true Windows kernel**
  - Can support applications *and* drivers

Wine on ReactOS



Services

Windows applications

System libraries

kernel32

ntdll

user32

gdi32

DirectX

User mode

Kernel mode

Executive

win32k

HAL

Kernel

Windows drivers

Wine and ReactOS: summary

- Wine was designed to run Windows applications on Linux. Linux-specific dependencies are:
  - … invisible to applications
  - … an integral part of Wine design
- ReactOS was designed to *be* Windows:
  - Needs to take as much as possible from Wine
  - Needs to reimplement what Wine implements in a Linux-dependant way
  - Implementation cannot just be “functionally equivalent”: must be “binary-compatible”, because in Windows *everything* is an API
  - **Not a lot of code can be reused from other projects**
- **ReactOS is complex and irreplaceable**

ReactOS is (not) Windows

REACTOS ARCHITECTURE

ReactOS *is not* Windows

- All the parts of Windows that aren't in Wine must be reimplemented
- This means the kernel and all kernel mode subsystems (graphics, sound, USB…). It sounds hard and it is
- Who was ~~stupid~~ brave enough to take this task, and did they succeed?

The ReactOS crew

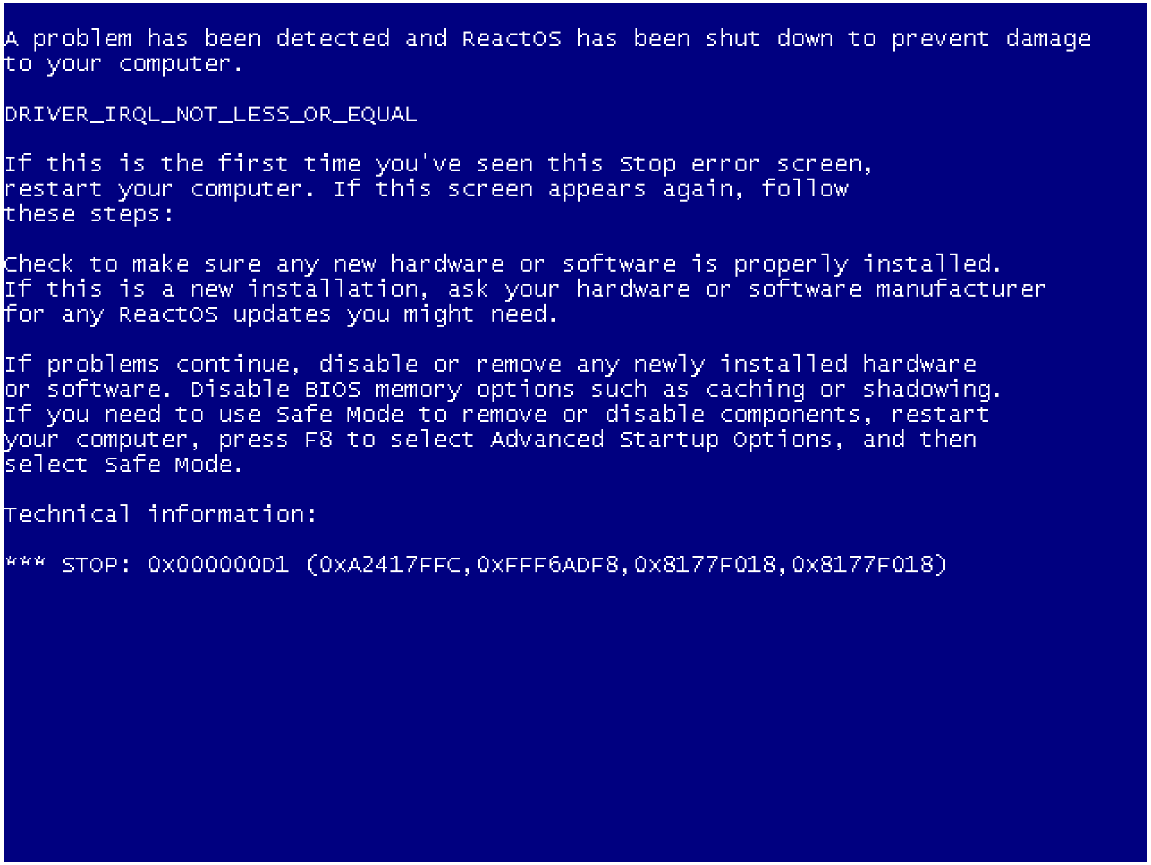
- A truly “international” project
  - Founded by Jason Filby from South Africa
  - Most early developers and the first ReactOS foundation from the USA
  - Today, a Russian foundation and project coordinator, most developers from Germany and the USA, and a community spanning the globe
- No formal training
  - Almost all developers learned Windows internals while working on ReactOS
  - Sadly for the project (but happily for them), the best developers are “snagged” by Microsoft and other large companies
- Very little information available to the public
  - “**Inside Microsoft Windows**” is *the* reference on Windows design and internals
  - … but it’s not enough information for ReactOS development

The ReactOS kernel

- Many developers alternated developing the ReactOS kernel and subsystem, with mixed results
- **Good** quality:
  - Scheduler, HAL, process and thread manager (thanks Alex Ionescu!)
- **Fair** quality:
  - I/O subsystem, configuration manager (registry), security manager
    - Security manager is good enough to support a prototype implementation of Mandatory Access Control (MAC) I did for my BS thesis
- **Poor** quality:
  - Memory manager, cache manager, filesystem support library: three tightly coupled components that have been our “white whale” since the beginning
- Non-existing:
  - Power management
- Nevertheless, the ReactOS kernel is…

The ReactOS kernel

- … compatible enough!



```
A problem has been detected and ReactOS has been shut down to prevent damage
to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any ReactOS updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0xA2417FFC,0xFFF6ADF8,0x8177F018,0x8177F018)
```

The windowing subsystem (USER)

- Another long-time “white whale” sub-project
- Many developers tried and failed
  - Three separate rewrites, one still ongoing
- The original implementation is **a very good hack**…
  - Windowing subsystem comes all the way back from Windows 1.0
  - The port to Windows NT introduced memory protection, but the API implies shared memory
  - Several dedicated hacks to simulate shared memory safely – user32.dll is not just a library, but the user-mode half of the windowing system
- … but **a really poor design**
  - Impossible to give an good, high-level description of the architecture
  - Nobody documents all of it, neither officially nor unofficially

The graphics subsystem (GDI)

- Tightly coupled with the windowing subsystem
- Much simpler, better design
  - gdi32.dll is a partial user-mode reimplementation of the subsystem, to run user-mode display drivers (i.e. printer drivers)
  - Drawing algorithms are well isolated in a simple API
    - All our font drawing code comes from FreeType (a third-party, open source project)
- Efforts concentrate on the more complex (and visible) windowing subsystem, however
- DirectX graphics is a whole another matter entirely…

Networking

- The networking stack in Windows is outside the kernel
- … but the stack is split into independent layers, with many documented APIs between them:
  - Winsock
  - TDI
  - NDIS
- Each part has to be implemented in a Windows-compatible way
- … but many parts are complex enough inside to make it possible to wrap a large third-party implementation in a Windows-compatible “shell”
  - Our TCP/IP driver is almost 100% FreeBSD code
- “Good enough” quality

What are we working on, what we will work on

REACTOS PRESENT AND FUTURE

Driver support

- Stand-alone drivers run well enough
  - Video drivers
- Complex abstraction layers need more work
  - USB
  - Sound
  - Network card drivers (except PCI Ethernet cards)
- Filesystem drivers (including network filesystems) require a lot of work on the kernel “big three” (cache manager, memory manager, filesystem support)

USB

- We used to use a port of the Linux Cromwell stack, but it “bit-rotted”
  - Used in the XBox port (XBox only supports USB input)
- We currently use a **USB compatibility layer for Windows NT 4**
  - “Good enough” for light use (USB keyboards, mice, etc.)
  - Windows NT 4 lacked kernel features to properly support USB, so the compatibility layer is very different from “real” USB support
- Our I/O subsystem is not ready yet for full, “real” USB support

Audio subsystem

- **It works!**
  - ReactOS can play audio
  - The audio subsystem prototype successfully played several hours of streamed MP3 audio through Winamp
- … but it’s very incomplete
- Hard to find people with experience in Windows audio

Kernel subsystems

- **Cache manager** rewrite is in progress
- The ARM port resulted in a large cleanup of the **memory manager**
- Overall quality improvements

Development tools

- We don’t support the Windows kernel debugger… **yet**
- We only support compilation with gcc, which doesn’t play nice with Windows tools
  - We contribute to the development of the Windows port of gcc (MinGW) because we are probably its largest user (and we find a lot of bugs in it!)
  - MinGW was never expected to compile a kernel!
  - I’m working on a build environment and source code clean-up to support compilation with Microsoft Visual C++
    - More accessible to new developers
    - Better integration with Windows development tools

What did we learn today?

CLOSING REMARKS

Summary

- Windows is a pretty normal operating system, after all!
- ReactOS…
  - … is (not) Windows: it’s a 100% open source reimplementation of Windows
  - … is not Linux: it runs Windows drivers
  - … is not Wine: it uses Wine, but Wine is only part of it
- **ReactOS is complex and unique**
- ReactOS is a lot of work

Any questions?

Brought to you by

REACTOS®

CREW