The Reactos Project

An Open Source OS Platform for Learning

Speaker Info

- Alex Ionescu
- Lead Kernel Developer for ReactOS Project. Have been working on the project for almost 3 years.
- Software Engineering Student in Montreal, Quebec (Concordia) and Technical Microsoft Student Ambassador.

Outline

- About the Project
  - Description
  - Motivation and Goals
  - Current Status
- ReactOS Architecture
  - Kernel
  - Native + Subsystems
  - User (Win32)
- ReactOS for Academia
  - The OS Perspective
  - The Software Engineering Platform
  - Google Summer of Code 2007
- Roadmap for 2007
- Getting Involved

Implementation, Motivation, Goals, and Current Status

ABOUT REACTOS

Description

- ReactOS is an operating system written from scratch.
- It is an NT-based kernel and closely follows NT architecture.
- NT is a 32-bit Windows-family OS written in the early 90ies by Microsoft and constantly updated by new releases. Windows 2000, XP, 2003, Vista are different versions of NT.
- ReactOS targets Windows XP/2003 (NT 5.1/5.2).
- ReactOS has been in development for 10 years, some code is based on NT 4 architecture, while some APIs support extensions added by NT 6 (Vista).
- ReactOS includes the kernel, Win32 libraries, system libraries and drivers, base applications, system components, subsystem support and window manager.
- ReactOS excludes anything not part of an NT installation.

License and Shared Code

- ReactOS is GPL 2.0 licensed, but it includes 3rd-party code under its respective compatible licenses.
- 3rd party code includes:
  - Wine makes up the bulk of ReactOS's Win32 Libraries, which are mostly left untouched.
  - Freetype provides font rendering support for the window manager.
  - libxml, libpng, bzlib, adns provide support for specialized Win32 libraries and applications.
  - MESA provides software OpenGL rendering.
- In return, ReactOS code has been used by:
  - Captive, for NTFS write support in Linux.
  - Some patches went upstream into Wine.
  - NDISWrapper.
  - LinuxBIOS support for booting NT.

More on Cooperation

- Due to the large size of our codebase and Win32 requirements, several patches and improvements have been made by our developers:
  - MinGW DDK Headers.
  - Patches to MinGW GCC and Binutils.
  - Patches to QEmu and KQEmu.
- As we begin supporting more Win32 specialized APIs, more libraries will probably by imported, especially for security APIs and GDI+ support.
- However, we will never include whole applications or APIs not present in Windows.

Motivation

- NT 5.2 provides a rich and extensible architecture with a highly scalable and optimized set of components.
- A secure and reliable OS, written for C2 security level certification, and updated to B1 for Vista.
- Also a great learning opportunity and research material for students and academia.
- Runs 95%+ of all computer software and drivers, most pervasive consumer OS on the planet.
- But…

Motivation

- Plagued by bad design decisions made early-on in 16-bit Windows 9x history but kept for compatibility.
- Plagued by a myriad of hacks to support badly written applications and drivers from 3rd party developers.
- Plagued by bad design decisions still being made to maintain corporate agenda (DRM, Driver Signing, etc).
- Plagued by bugs in bundled software (Internet Explorer/Windows Media Player/Outlook Express) and bad security decisions (users run as Administrators, etc) which undermined architectural security and reliabilty.
- Closed source, costly, poorly documented in regards to system architecture and undocumented functionality (compared to competing FOSS operating systems).
- Most extensibility features kept undocumented and not open to 3rd party modification.

Goals

- ReactOS aims to offer all of the features and performance of NT without all the hacks, restrictive design decisions and license restrictions.
- It aims to offer no-cost Windows compatibility at a level no other solution can.
- It aims to document the undocumented, and to provide binary-compatible components which would be used to provide extensibility.
- Great teaching platform for academia. UNIX/Linux are good to learn from, but NT does some interesting and different things that deserve the same attention.
- Will not include applications such as IE, OE or WMP. Users will be encouraged to install FireFox, ThunderBird, OpenOffice, Mplayer, etc.

Current Status

- Large parts of the kernel are now fully compatible with Windows 2003 SP1: Executive, Kernel Core (Scheduling, Dispatching, Interrupts, etc), HAL, Local Procedure Call, Process and Thread Management, and most of I/O support (except PnP).
- Other parts are totally foreign compared to NT design, notable the Cache Controller, Configuration Manager (Registry backend) and Memory Manager.
- Win32 application support largely depends on two components:
  - Win32k – Kernel-mode GUI Server, analogous to X.
  - Win32 libraries (gdi32, user32, kernel32, advapi32) – Taken from Wine.
- Some Win32 functionality depends on kernel behavior.

Current Status

- User-mode applications are currently supported in a limited fashion – specific apps targeted: Firefox, Thunderbird, OpenOffice, etc.
- 90% of application compatibility problems are due to Win32K or subtle kernel bugs.
- Theoretically, ReactOS should run at least what Wine runs.
- Drivers have not been fully tested, but several problems exist:
  - Some drivers use hacks that depend on offsets, memory locations and variables specific to NT.
  - Other drivers depend on subtle PnP implementation differences (synchronous vs asynchronous behavior).
  - File-system support is bad due to deep Memory Manager/Cache Controller differences.
- On the other hand, the Windows 2003 USB Stack works nearly perfectly out of the box, as does, for example, the Named Pipe File System driver.
- NVidia video drivers hack deep into kernel variables, so they need binary patching.

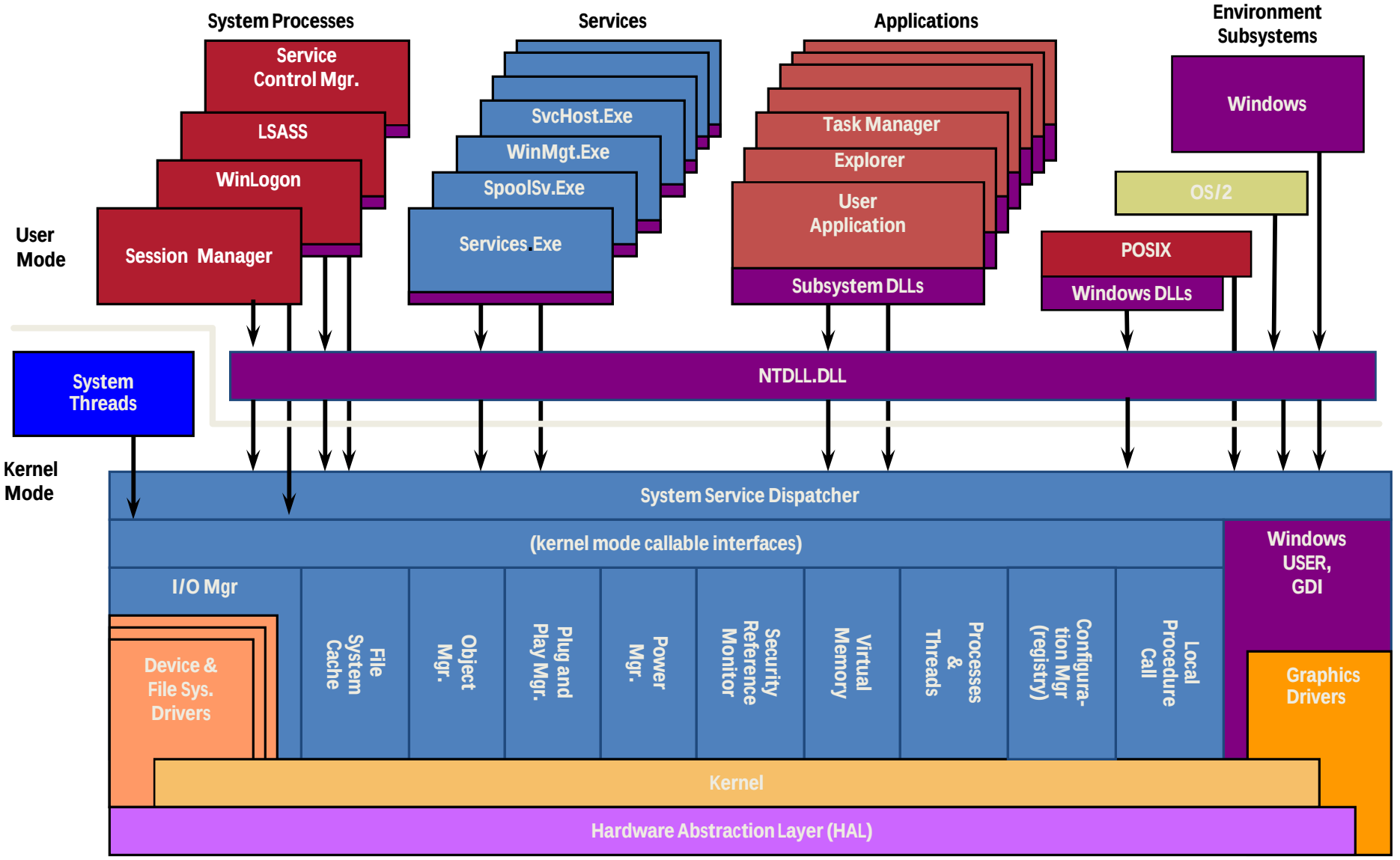
Kernel, Native and User Mode – Subsystem Paradigm

REACTOS ARCHITECTURE

NT Kernel Mode Design

- Kernel-mode NT is implemented by one large module (ntoskrnl), a hardware abstraction layer (HAL), and a set of loadable kernel modules (drivers and other kernel libraries).
- The kernel is written in portable C, and ports exist (or once existed) for MIPS, Alpha, AXP64, Sparc, i810, IA64, x86, x86-64, PowerPC. Some low-level parts are written in assembly for each architecture.
- Therefore a HAL is required to manage hardware-specific implementations: interrupts, processor initialization, DMA, PCI/ISA bus access, timers, etc.
- Some drivers are specialized for a type of hardware (PCI driver, ATAPI driver, IDE driver, etc).
- Other drivers are generic modules, or file systems (Partition Manager, Volume Manager, NTFS, Mailslot Driver, Kernel Debugger Library, etc).

Windows Architecture



ReactOS Kernel Mode Design

- Nearly identical to NT.
- Current tree has \ntoskrnl directory implementing the kernel itself. CONFIG_SMP switch exists, mimicking NT's NT_UP, to build an SMP-compatible kernel (due to some tiny differences in dispatching code related to synchronization – especially spinlocks).
- Also have a \hal directory, with \halx86 and \halxbox. \halx86 also has UP vs SMP specific code (spinlocks and interrupts are implemented in the HAL). Main difference here is that SMP machines have APICs, UP machines have a PIC, usually.
- No ACPI support yet – HAL uses PIC and legacy timer hardware.
- Currently the only port being worked on is PowerPC.

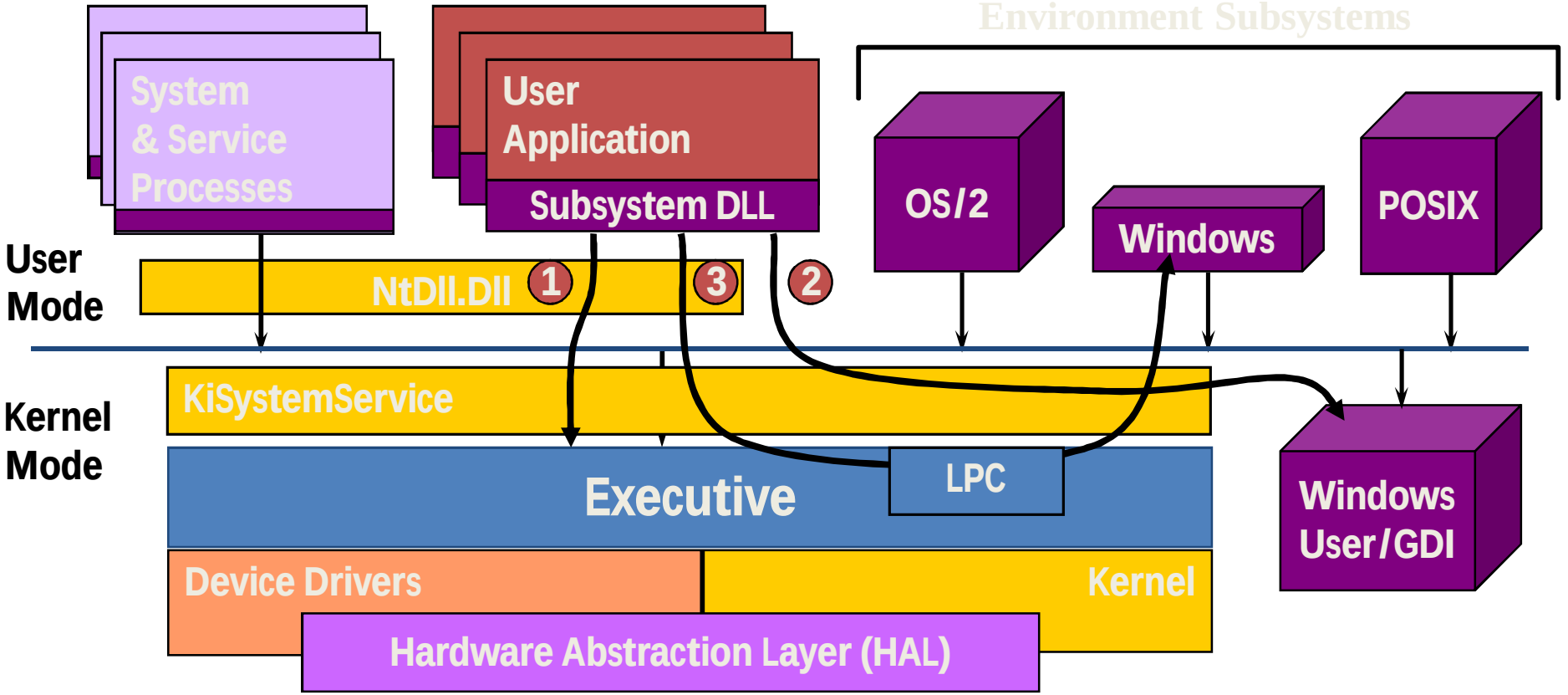
Kernel Design Differences

- As mentioned earlier, the Cc, Cm APIs are incompatible, even at the application level. This is a show stopper for many drivers.
- Mm APIs are more compatible, but internally implemented differently. This is currently hurting performance and stability.
- PnP Manager communicates with drivers through IRPs (I/O Request Packets) and offers notifications/events/operations in a specific order – ReactOS doesn't have this fully compatible yet.
- NT Boot Loader sets up paged mode, IDT, GDT, TSS and low-level system structures. ReactOS boot loader works in protected mode only, and the kernel is responsible for system structures. This is currently being addressed.

Native Design - Subsystems

- Microsoft wasn't originally sure of the target API that the NT platform would support; original plan was OS/2.
- This later changed to Win32; designers needed to find a way to support this change, as well as implement POSIX compatibility.
- NT Subsystem model was created, similar to BSD's architecture.
- NT exposes system calls (Native APIs) that are at the core of the kernel.
- One Session Manager process, written in "native" mode, loads subsystem processes.
- Subsystem processes register non-native applications, such as Win32 or POSIX, which come with their own DLLs.
- These DLLs wrap the Native APIs, and sometimes need to call the subsystem process for functionality not accessible/available through Native APIs.
- Win32 supports consoles, NT doesn't -> subsystem process handles this.

Windows Simplified Architecture



1. most Windows Kernel APIs
2. most Windows User and GDI APIs (these were formerly part of CSRSS)
3. a few Windows APIs

Native Design - Subsystems

- Some functionality cannot be emulated by subsystem processes in user-mode alone -> a kernel-mode subsystem server is further required.
- Case in point: NT doesn't natively support the concept of a GUI, so Win32K acts as the subsystem server for gdi32 and user32. Kernel32 on the other hand can wrap native APIs (except for Console API).
- The only DLL NT comes with is ntdll, which has the system call entrypoints for all the Native APIs, as well as a Runtime Library, with over 600 Rtl* APIs for various operations (similar to the C runtime library/standard library).
- NTDLL also contains the loader for all PE files, but it doesn't know how to register a non-native application with its subsystem server.
- Kernel32 does this, by looking at the subsystem header in the PE file, which identifies it as POSIX, OS/2, WinCE, WinNT, etc.

ReactOS Native Design

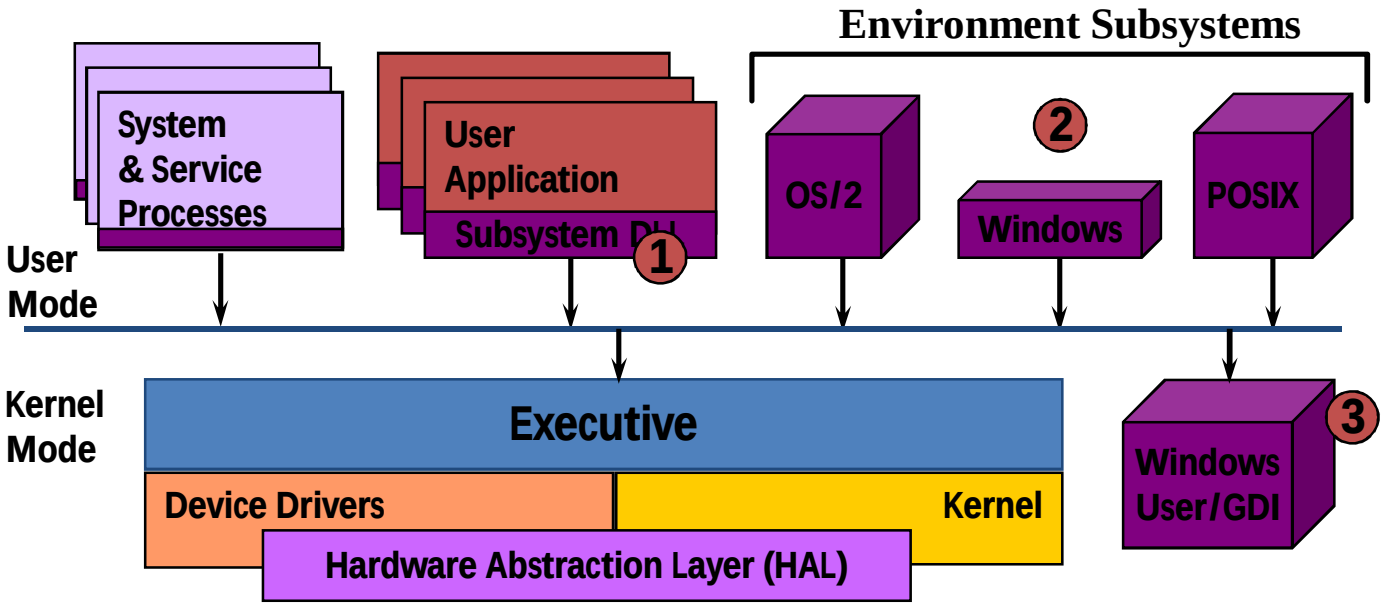
- Closely follows NT but allows more extendibility.
- Session Manager allows custom-made subsystems, and doesn't have any hardcoded assumptions like NT's
  - SkyOS subsystem was made to run SkyOS applications.
- Win32 subsystem server (Win32K) is present, as well as the client (csrss), however Win32K APIs are not compatible (not required to).
- CSRSS communication with Win32 DLLs is done differently, but this is also not really required from a compatibility standpoint.

Win32 User Mode Design on NT

- Base API located in kernel32. Handles subsystem-part of loading applications and registering with subsystem client.
- Console API is also located in kernel32, talks with subsystem client for functionality.
- Graphical functionality is located in user32 and gdi32, which talk directly with the subsystem server through system calls.
- Other DLLs wrap more complex functionality directly in user-mode, or by calling kernel32 (advapi32, netapi32, winsock, etc).

Subsystem Components

1. API DLLs
   - for Windows: Kernel32.DLL, Gdi32.DLL, User32.DLL, etc.
2. Subsystem process
   - for Windows: CSRSS.EXE (Client Server Runtime SubSystem)
3. For Windows only: kernel-mode GDI code
   - Win32K.SYS - (this code was formerly part of CSRSS)



Win32 User Mode Design on ROS

- Can't use Wine for kernel32. Our implementation needs to call ntdll, which then performs the system calls, while Wine implements the APIs directly in the DLL.
- Same principle applies for GDI and USER32 libraries. NT design dictates going through kernel-mode (Win32K) – Wine does this inline.
- Other DLLs are directly shared from Wine with minimal changes to support our headers and build system.
- DLLs are supposed to work and be binary compatible with Windows.

Learning with, and from, ReactOS

REACTOS AND ACADEMIA

The OS Perspective

- Why is learning about Operating Systems important?
  - Complex challenges.
  - Extreme code coverage.
  - No assumptions.
  - Optimizations are a double-edged sword.
  - Gateway to user's data, privacy and security.
  - Reentrancy and concurrency.
  - Access to all of machine's hardware.
  - Used on everything from critical live-saving devices to cars to cell phones.

Complex Challenges in an OS

- Defining the number of services to expose can be time-consuming. How many APIs to write? What should be up to 3rd party developers? What should be native to the kernel? What should go into user-mode?
- An OS can be deployed on millions of machines very quickly, but patching can take months/years. Sometimes an OS is on a ROM – patching is unfeasible. One mistake can kill your OS in the market.
- Dependencies and backwards compatibility. Eventually, you will have to change things… will the application base from 3rd party developers break down?

Complex Challenges in an OS

- Communicating with the industry: hundreds of thousands of applications may have been written for your OS. Do you keep in touch with the developers, or is your OS a free-for-all?
- Hardware changes. Your CPU architecture may become obsolete (eg. Apple PPC). Is your code portable? Did you properly abstract away hardware implementation details?

Extreme Code Coverage

- Code coverage means: how much of the code that you've written is actually executed. A large function can support 15 flags and 10 code paths which execute different code. A normal app might only use 15% of that code.
- In an OS, likely 90%+ will be used at some time or another, due to the sheer amount of applications.
- Profilers and tracers will usually only show you the most typical code paths: you need to write test cases!

Extreme Code Coverage

- But even your test cases are done by *you.* You might miss bizarre behavior or calling contexts.
- It's hard to optimize code that has near-100% coverage (more on that later).
- Finding bugs is also hard: race conditions might only happen in one path of your code, and only when called from within another of your functions.
- Are you going to support those three applications that use a flag nobody else does, or remove the flag and optimize the routine?

No Assumptions

- Making assumptions when writing an OS will make your code worthless.
- As mentioned, code usage makes it hard to assume things: everyone is using all your code.
- There are still cases when some flags/routines are severely underused. Can something be done?

No Assumptions

- Yes! Large-scale profiling and tracing (instrumentation). Microsoft case study:
  - Checked Build of the OS for developers. Ships with “assertions”. Assumptions the developers are making, which, if violated, will crash the OS with the code actually printed. Eg: “Assertion failed: Thread->State == Running. Line 285 File ntos\ke\thredsup.c
  - Code Coverage builds of Server 2003: Instrumented routines that logged all calls and flags and calling contexts, then sent the data back for analysis.
  - Internal test labs of up to 10 000 machines, with various configurations.
  - Events such as “Plug Fests”, where 3rd party developers bring in their drivers and plug them into new kernel functionality. Usually result in multiple kernel bugs being found.
  - Events such as Vista Compatibility Workshops, where developers brought their apps and tested them under Vista, and received help from actual Microsoft developers.

Optimizations

- Optimizations can be generic (eg. Using a merge sort to sort a linked list) but this is now rarely the case.
- Specific optimizations are usually done by improving one path of the code, for one (or maybe a couple) of specific usages.
- But again, an OS has all the code coverage problem.
- You can optimize your heap algorithm to be low-fragmentation, and this will provide large speed increases to applications making repeated, small allocations.
- However it will slow down applications making large allocations.
- An OS will have both kinds of applications. What if large allocations are only made by 3% of applications? What if one of those applications is made by a partner ISV with 200 000 users in the largest financial organizations? What if applications are 50/50 on heap usage?

Optimizations

- Instrumentation will give you some answers related to usage patterns, but the economical aspect is an unfortunate aspect of commercial development.
- Decreasing performance for a key ISP might mean hundreds of millions in lost revenues and a switch to Linux.
- Therefore, you need to optimize *all* cases and use **dynamic code.**

Optimizations

- Eg: Low-Fragmentation Heap added in Server 2003. Can be enabled by the application at run-time, or by the developer at compile-time. The OS doesn't choose, the developer does. A simple but not very elegant solution for legacy applications.
- In Vista, however, the OS now has algorithms to detect your usage. The OS actually profiles the application, and determines if the LFH would help. Perfect engineering solution, but highly complex to implement… and this is only one scenario!
- Apple is only starting to take this seriously.

Optimizations

- On the other hand, some more generic optimizations can provide large performance improvements.
- Windows XP added Pushlocks, a pointer-sized reader-writer lock (patented). Uses simple CPU interlocked operations for locking/unlocking. Waits and large code only done in contention cases.
- Previous lock implementation used 50 bytes per lock, and required complex API routines just to check the state of the lock and acquire it.
- Handle implementation was rewritten in part to use pushlocks.
- Result: 30% improvement in handle usage, and since almost all of Windows relies on handles, this is one of the most visible (to the user) improvements between 2000 and XP.

Trust, Privacy, Reliability

- Word is responsible for not losing your documents, and IE keeps your web privacy safe. But not vice-versa; as a developer, you don't need to worry about someone else's data files.
- As an OS, you do. Every read/write command goes through you; crash, and the entire disk might become unusable.
- When an OS crashes, the system goes down, the state is unsaved, data is lost, and hardware damage can occur. When an application crashes, only the local data is lost.
- Applications should *never* make the OS crash (Windows 9x anyone?)

Trust, Privacy, Reliability

- It's very hard for Word to get your credit card numbers stored in Firefox, because of memory address space protection and user credentials.
- It's trivial for code running in kernel-mode: there's no "user", there's no memory protection. If the OS has a bug which allows malicious kernel code to execute, your entire computer becomes unsafe.
- When such a bug can be exploited through a web page or remotely through an open port, things get really bad.

Trust, Privacy, Reliability

- Rootkits: Malware which installs itself in kernel-mode and hides information from user-mode, or lies. Remember that almost every user-mode API needs to call the kernel. What if the kernel returns invalid data? The application has no (simple) way of knowing/detecting this.
- How can an OS protect against rootkits? Microsoft uses PatchGuard in Vista/2003 64-bit editions, but paid a heavy price with 3rd-party developers. In 64-bit Vista, kernel-mode drivers need to be signed by Verisign, which caused even greater unrest in the community.

Trust, Privacy, Reliability

- In recent server environments, an OS can be running 16 instances of VMWare ESX Server, which emulate 16 different servers: Web, DNS, Email, Terminal Services, etc. If the OS on the host machine crashes, that's 16 servers down.
- All this to say that your OS code, especially in kernel-mode, needs to be nearly perfect, and when mistakes happen, they're usually pretty bad.

Concurrency Issues

- An OS needs to be ready to handle SMP (Multi-Processor) machines. In the past this wasn't such a big issue except on high-end machines, but dual-core is now becoming pervasive.
- A really good scheduler needs to differentiate between true SMP (multi-core) and SMT (HyperThreading). The latest scheduler algorithms can even tell between multi-core and multi-chip.
- NUMA: Non-Uniform Memory Access. On very large servers, the 2GB of memory that an application needs might be near CPU 16. CPU 16 might have 15% load, and CPU 54 might be idle, but not as close to memory. Up until recently, most schedulers would pick CPU 54.
- Windows 2003 has a fully SMT, SMP, NUMA compatible O(1) scheduler.

Concurrency Issues

- Your application may only be using one thread, since you don’t want the complexity of multi-threading, or don’t need it.
- The OS uses dozens of threads, and also has to manage yours. Ever since the first versions of NT, this was an issue.
- In NT, threads can be pre-empted even on single-processor machines, so race conditions can still occur if proper mechanisms are not used.

Concurrency Issues

- Imagine the following piece of code:
  - *Pointer++;
    if (*Pointer == 4) doSomething();
- Suppose the value before increment was 3. What if the thread gets pre-empted before the condition, but after the increment? A new thread could be scheduled, and run the same routine again, incrementing the pointer to 5, and failing the check. Back to the original thread, the value is now 5, also failing the check. doSomething(); never executes!
- Another thread could be scheduled in different code, which also touches this pointer, if the pointer is a global variable accessible to other modules.
- On 64-bit SMP machines, such as an IA64 (Itanium), even the act of reading the pointer can cause a race condition! Specialized C macros exist to protect for this.

Concurrency Issues

- Solutions?
  - Synchronization objects. NT exposes mutex, fast mutex, spinlock, resource, pushlock, fast reference, rundown, semaphore, event, timer, queued spinlock, IRQL levels as various methods of protection.
  - Some are generic, others are specific for a special kind of problem. Some are CPU-level locks, others are OS-implemented and allow multi-reader-single-writer locks, etc.
  - Caching values when reading from a pointer to make sure the original value was kept.
  - Using interlocked APIs/macros, such as InterlockedIncrement. The operation is guaranteed atomic and the previous result is returned.
  - InterlockedCompareExchange allows most locking code to function properly on x86 CPUs.

Direct Hardware Access

- Unlike applications, the kernel of an OS runs in Ring 0, which means it has direct access to hardware.
- Interrupts and Port Access, DMA access is all possible.
- The OS can flash the BIOS, and even do CPU Microcode Updates.
- Recent CPUs support “MSRs”, or Machine Specific Registers, which can control everything from debugging features to power throttling (it would be trivial to burn a CPU by disabling the fan and power-saving features, then looping at 100% CPU usage, on a laptop).
- Not sending the right commands to a piece of hardware could irreparably damage it.

Direct Hardware Access

- Hardware access isn't only about damage or risk, but also a programming problem.
- Some technologies are standardized: IDE, PCI, ATAPI...
- Vendors don't always follow them. VESA (for video cards) has over a dozen different implementation differences between various video card manufacturers.
- The Keyboard Driver for PS/2 in Windows NT has ten hacks just for different things like NEC Japanese keyboards. The Disk driver has an entire *table* for various manufacturers.

Finally…

- As a student, working on an OS exposes you to all those issues, and more.
- People that can not only understand those issues, but find creative solutions to them have their career set ahead of them.
- Average developers may have never even bothered about multi-threading until it hits them in 2010 and their code breaks down.
- In today's scalable and maintainable world, code monkeys don't cut it anymore.
- Once you've worked on an OS, you can pretty much work on **any** other software project.

ReactOS as a SOEN Platform

- The challenges behind the development of an operating system are not limited to technical/code challenges.
- A build of NT takes a whole night on more then a hundred machines in a build farm. ReactOS can take up to 2 hours on an average low-end P4, but we only have 1% of NT's codebase.
- Some headers are used by every single component in the source tree. One structure change can break a very unrelated small command-line application.
- Source tree is so large that the repository is de-synchronized, sometimes up to months (ReactOS doesn't do this).
- An OS project requires people, lots of people. How do you manage such a large development team?
- ReactOS uses 3rd-party build compilers and libraries. How do you handle bugs in those components which you don't own?
- ReactOS can be built on both Windows and Linux, supports 4 different emulators, and lots of hardware. How do you test?

Building Issues

- A build can take a lot of time for a simple change such as fixing the name of a member in a public structure, since the entire tree must be rebuilt. The last application in the build list may still be using the old name. You just wasted 2 hours.
- You're lazy and commit immediately. You've just wasted 2*n users/developers hours and broke the build for everyone.
- Testing, testing, testing!

More Building Issues!

- GCC is currently our compiler, but some people want to use MSVC, CodeBlocks, Dev C++. We have over 2000 makefiles, how to support all those other environments?
- OS needs to be built from scratch. Notepad requires kernel32. Kernel32 requires ntdll. Ntdll requires ntoskrnl.exe. Ntoskrnl.exe requires the CRT and other libraries.
- The CRT is mother, the CRT is father. It is the first component that must be built, and all other libraries are either independent and build on that. Once ntoskrnl and ntdll are ready, drivers and user-mode applications can start compiling.
- How to handle these dependencies? Do you keep a list? What if a new developer adds some app?

Some Building Solutions

- Create a building platform with a backend and frontend. Microsoft uses Build/Dazzle. ReactOS uses “rbuild”.
- rbuild does dependency checking, and uses a native .xml makefile format. Each development environment has an rbuild frontend, which converts the .xml files to a 3rd party format (makefile, .vcproj, etc).
- Interesting chicken-and-egg problem: To build the OS, you need rbuild… but to build, you need…rbuild. (We don’t ship any binaries, it must be built from source).
- rbuild is fully cross-platform (Linux/Windows).

Build Breakage

- Because of rbuild, if a header changes, rbuild can detect which applications were using it, and only rebuild *those* applications: saving compile time.
- If a broken commit is still made, how do you protect against wasting everyone's time? BuildBot!
- BuildBot is an open source project that monitors commits and starts up the build environment for the project. It comes with a web interface and various monitoring utilities (including an IRC bot).
- To be useful, BuildBot needs to be **FAST!**

Build Breakage

- Since last week, Build Server is now running on an Dual Quad-Core Xeon machine with 4GB of FB-DIMM DDR2. Plans set in motion for ~4GB of flash memory for instant disk access when building.
- A build should hopefully take < 3 minutes when the system is ready, giving enough time to catch any build errors before people start building.
- But what about Regression Testing?

Regression Testing

- A commit may build fine, but break a subtle part of the OS. We don't ship with 3rd party applications, so maybe FireFox is broken. How to detect that? Maybe the 2nd-stage installer broke on ASUS Motherboards.
- SysReg is a new utility being worked on that allows automatic regression testing under a variety of emulators (not hardware, unfortunately).
- Will automatically use an .ISO from BuildBot and attempt installation, setup and boot to desktop, as well as downloading, installing and executing various applications in our test suite.

Interaction with other projects

- A bug in zlib (unlikely!) might be breaking your installer. The zlib developers might have dropped the project, or maybe a bug fix will take a couple of weeks. How do you handle 3rd party code in the first place?
- SVN Vendor Drops! Allow you to “drop” a vendor library in a special part of the repository, and then “import” it into your source tree. You can then make local modifications to your copy (such as a non-official fix), but still have the official vendor drop.
- When a fix is available, you do a new drop, and import the official version.
- Vendor dropping is essential to our imports of Wine code, in which we often find multiple bugs/incompatibilities with the way NT does things.

Interaction with Wine

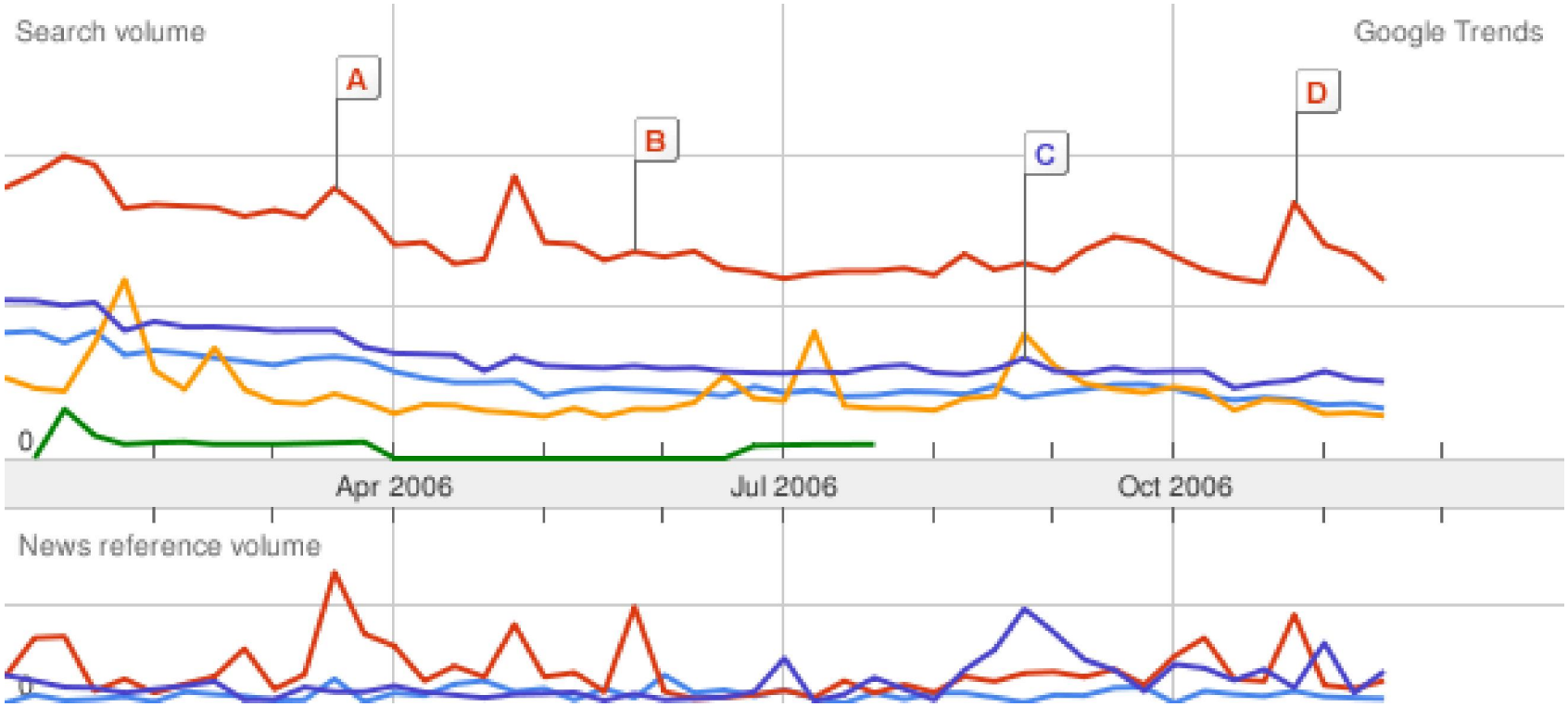
- Because Wine will never merge some of our changes (for multiple reasons), we always need to have a modified version of the drop. This can be time consuming.
- Wine also uses a different build system, with their own hacked set of incorrect headers. We use PSDK/DDK compatible headers.
- Solution: wine2ros.diff. An automated SVN bot automatically does periodic vendor drops of Wine code, and reads a human-generated diff file that is generic. In 99% of cases, the proper ROS-compatible rbuild.xml file is generated, headers are modified, and local code is patched.
- I keep bringing up “after a commit”. How?

SVN and CIA

- SVN allows pre and post-commit hooks, which allow a variety of powerful add-ons to the build system.
- Post-commit hooks we use:
  - ros-diffs mailing list mailer: Generates an email message for each commit.
  - CIA: notifies the IRC channel of a new commit, with appropriate information, and adds a new entry to the CIA repository.
  - BuildBot: Tells it to start a new build.
  - More internal hooks for various usages.

Development Community

- ReactOS has thousands of users around the world.



Development Community

- Dozens of developers from around the world contribute to ReactOS, with different time zones, languages and interests.
- IRC is our main development hub. Channel regularly has 120+ visitors, including developers, testers, translators, enthusiasts and trolls.
- Website receives tens of thousands of hits each day and provides rich services such as a forum, bugzilla database, wikipedia, application compatibility database, CMS, online translation utility, etc.

Development Community

- Managing such a community requires more then smart developers, and must include people from a variety of backgrounds and with different interests.
- It's hard to set hard goals like "Implement Sound support". Many developers might have no idea how to do this, and may want to work on their own stuff.
- But without goals, the OS expands like an octopus with many legs in every direction, but no clear goal.
- ReactOS is currently suffering a bit from this problem as well as major regressions and instability issues. 0.3.1 has been delayed by months already, and trunk is full of rich new features, but only boots on Qemu.
- Good PR and media people are required, as well as intelligent testers, media artists, translators, webmasters and moderators to keep the whole wheel turning.

Teaching and Learning ReactOS

- Now that you've seen the excitement and challenges behind writing an OS, maybe you're interested in ReactOS/NT and how to get involved.
- It's also a great teaching platform, from everything to scheduling, dispatching, inter-process communication to drawing routines.
- Start with these two resources:
  - Windows Internals, 4th Edition by Mark Russinovich.
  - Windows Curriculum Kit (CRK), available on MSDN Academic Repository (not MSDNAA, this one is public).

Google Summer of Code

- ReactOS will be having 4 (tentative) projects for this summer. Successful students will receive 4500$ USD from Google, as well as a T-Shirt and certificate. Each student will be mentored by a ReactOS developer and will receive adequate training to keep working on ReactOS after GSoC is over.
- The projects are:
  - Implementing PDB support for GCC.
  - Implementing SEH support for GCC.
  - Rewriting our Explorer clone from scratch, in C.
  - Writing a clone of the MMC (Microsoft Management Console).

Anticipated progress and usability

ROADMAP AND CALL TO ACTION

2007 Roadmap

- We’re hoping that the kernel will be entirely complete and compatible for 90% of drivers out there, by year’s end.
- Also hoping for Base/Core APIs to be fully working and regression tested.
- 2007 will feature heavy work on Memory Manager, Cache Controller, FS support, PnP Support, Win32K, DirectX, Sound support and Windows Networking.
- Largest focus will be on stability and usability however; 2006 suffered from too many rewrites and changes without proper testing and regression checks.

ReactOS and Google

- Google Summer of Code 2006 was a great success for us:
  - Clipboard support was implemented.
  - Terminal Services Client was implemented.
  - WinLogon project abandoned but picked up by another developer, so it was completed as well.
- Google fosters a variety of FOSS projects:
  - Wine
  - Subversion (SVN)
  - GCC
- ReactOS could be a very interesting platform for Google.

Call to Action

- ReactOS needs help!
  - Financial: Fundraising campaign is now in effect, hoping to raise 4000 Euros.
  - Manpower: Desperately require more developers familiar with Windows development, either Win32 or kernel mode.
  - Testers: Regressions need to be caught sooner and faster; automated systems aren't always perfect.
  - Awareness: Project is very popular in some European countries (notably Germany) but has poor penetration in US/Canada. We could reach out and help a lot of people!

Getting Involved

- Google Summer of Code 2007 would be a great way to help with some of the more involved/specific projects, and make money/experience during the summer.
- Writing a couple of good patches, being involved on the IRC channel/mailing list will grant you SVN commit access to the entire tree.
- Also looking for translators, artists, testers.

Software Projects

- User-mode Applications
  - Graphical (mmc.exe, syskey.exe, etc)
  - Command-line (at.exe, cipher.exe, etc)
- Kernel-mode Drivers
  - Storage stack (partmgr.sys, ftdisk.sys, etc)
  - Audio stack (wdmaudio.sys, ks.sys, etc)
  - Hardware drivers (NICs, Sound Cards, etc)
- APIs in DLLs, Kernel
  - Most user-mode DLLs come from Wine, but not:
    - gdi32, user32, kernel32.
  - Some Kernel APIs are still unimplemented: Wmi* for example.
- Build Tools
  - Dependency Map for rbuild
  - PDB, SEH support for GCC
  - Speedups/improvements for rbuild
  - Bind.exe clone with Linux support.

Questions and Comments

- http://www.reactos.org
- ReactOS IRC Channels on Freenode – #reactos, #reactos-dev
- ReactOS Mailing Lists – ros-dev, ros-diffs, ros-general
- My blog: http://www.alex-ionescu.com
- My email: alex.ionescu@reactos.org