Windows Kernel Internals II
System Extensions

University of Tokyo – July 2004

Dave Probert, Ph.D.

Advanced Operating Systems Group

Windows Core Operating Systems Division

Microsoft Corporation

Kernel Extension Mechanisms

I/O Extensions
- File System Filters
- New File Systems
- Device Filter Drivers
- Device Drivers

Object Manager
- *New object types*

Registry
- Hook most operations

Notifications
- Image Loading
- Process Create/Exit
- Thread Create/Exit

Export Drivers

Random bit editing

Kernel Communication

- IOCTLs
- Handles on new object types
- LPC
- Most usermode-to-usermode mechanisms
  - Shared memory
  - Kernel synchronization objects
  - NamedPipes

Kernel Extensions

Two main toolkits for writing extensions:

- IFSKit – for file system filters and file systems
- DDK – for all others, including device drivers

Generically called 'drivers' and use driver mechanisms to wire into the system

- DriverEntry routine creates a device object for the device
- Device object can be named in NT namespace
- Access via I/O ops (open/read/write/ioctl/close)

Service Control Manager loads/unloads drivers as 'services'

Published Kernel Interfaces

I/O related

- IO object mgmt, security checks
- HW access, DMA, interrupts, DPCs, timers, worker threads
- IRPs, physical memory (MDLs), cancel support (include CSQs)
- Hardware configuration, plug-and-play, power, bus mgmt

Multithreading support

- Spinlocks, interlocked operations/queues

Kernel facilities

- Memory pool allocation, threads, synchronization, run-time, object/handle management

Zw related (Kernel-mode version of native Nt APIs)

- Files, sections, registry, set/query file/process/thread info

Subsystems

NT originally mistaken for a microkernel

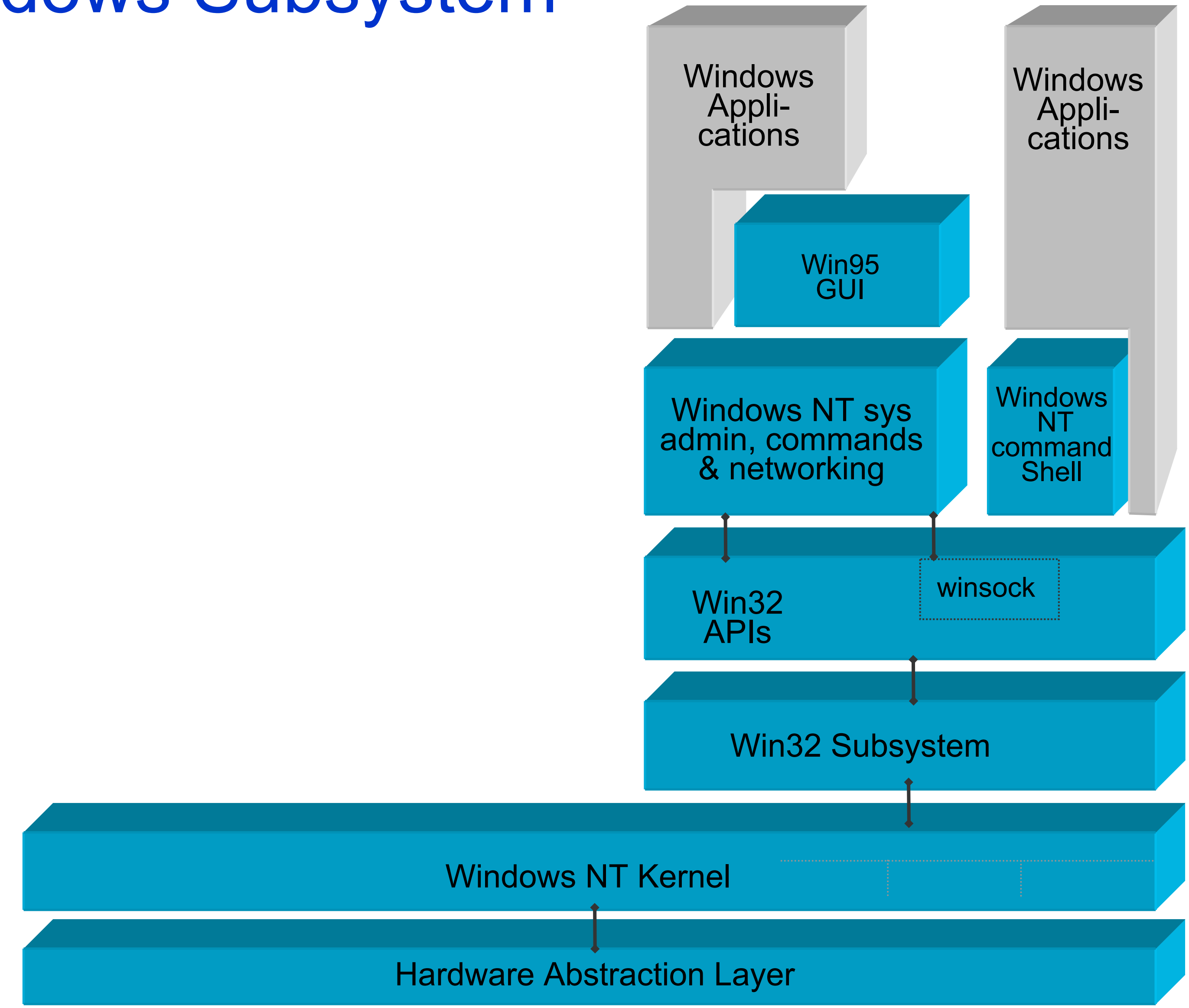
- Kernel was never micro, but …
- But OS personalities were defined by servers

Servers are called ‘subsystems’

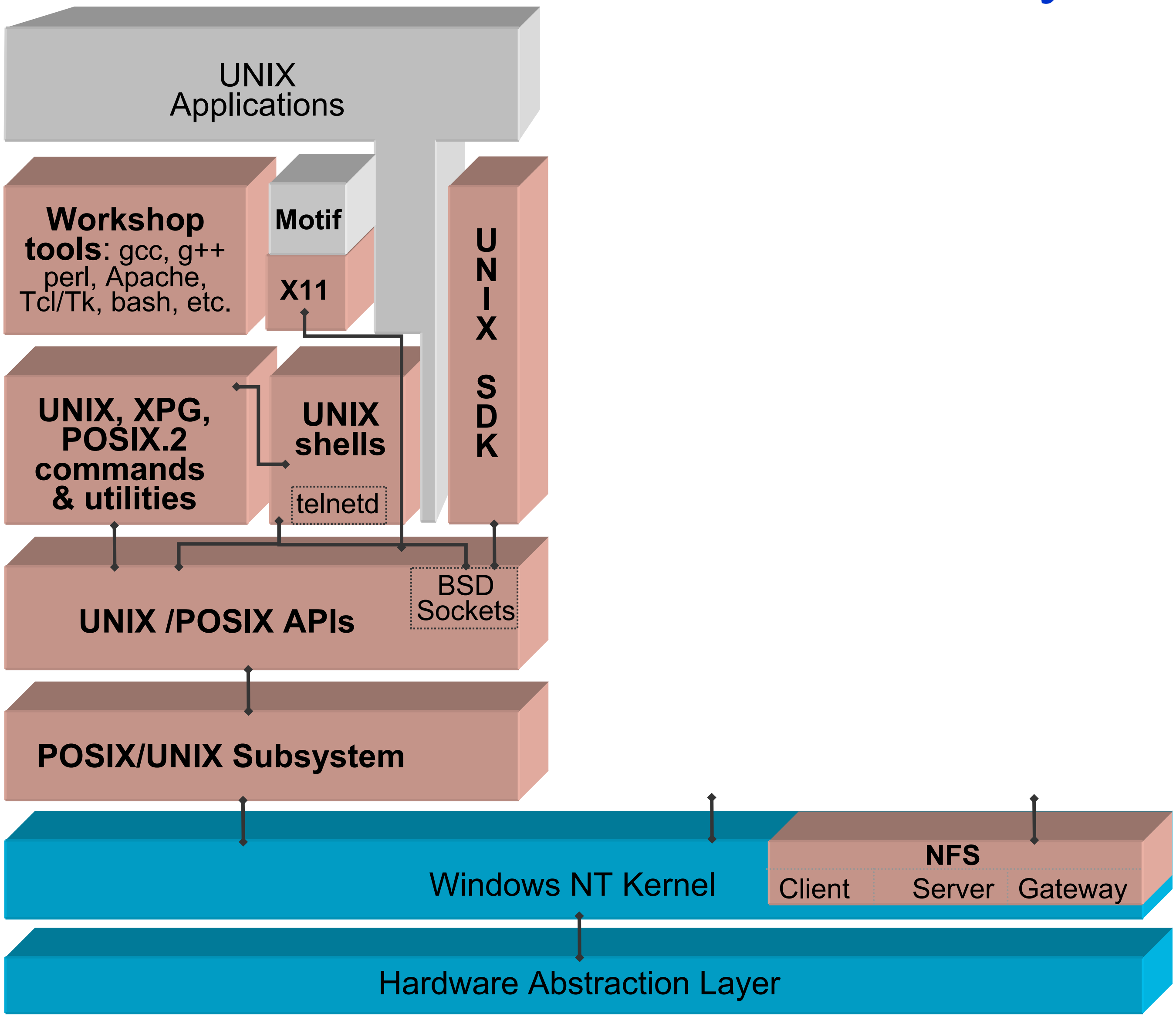
- Primary subsystems OS/2, Windows, Posix, WoW
- Each subsystem has three main components:
- Subsystem service process (e.g. csrss)
- Subsystem API library (e.g. kernel32, et al)
- Hooks in the CreateProcess code

There are some pseudo-subsystems, e.g. lsass, CLR

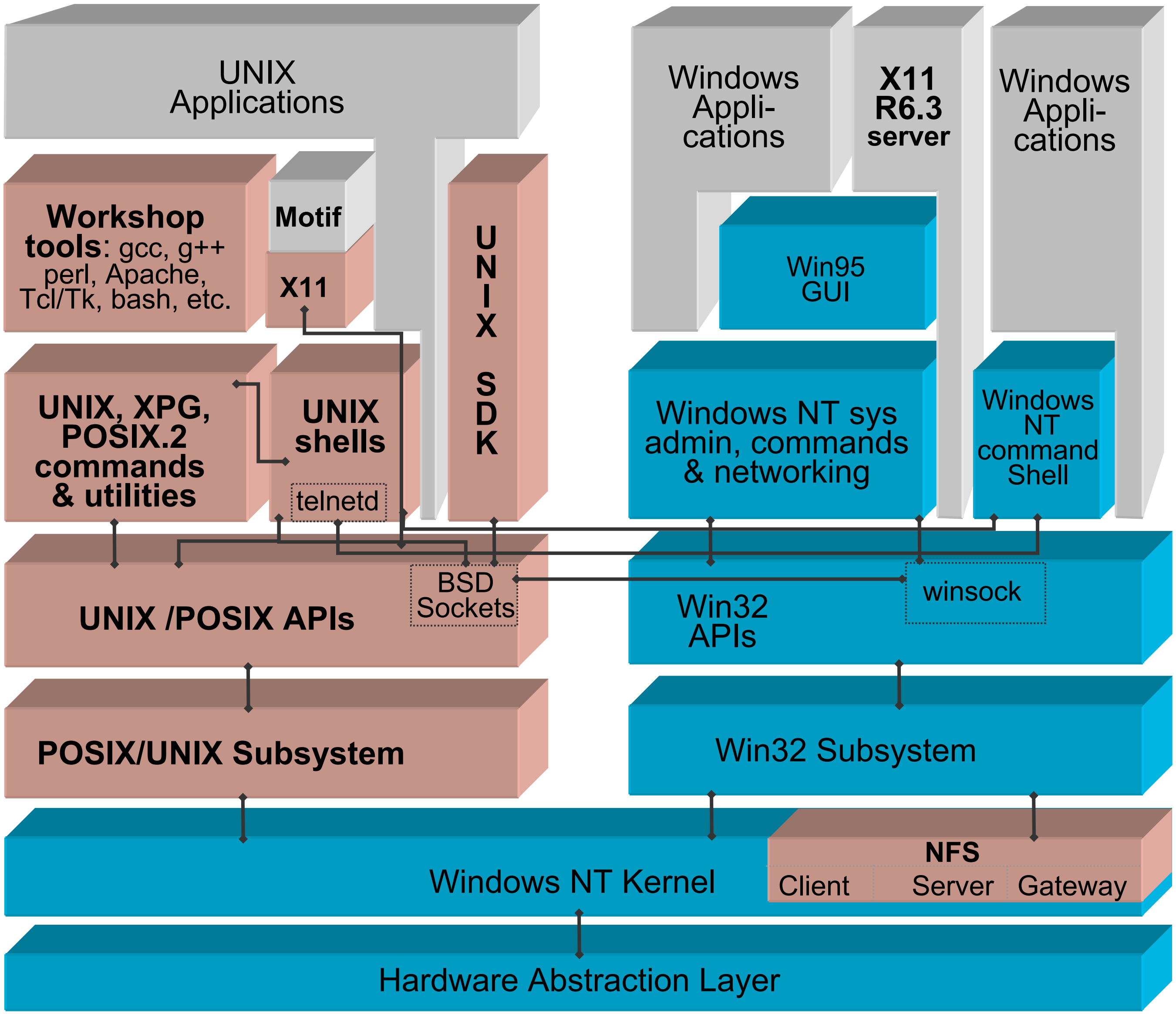
Windows Subsystem



Posix Subsystem



Subsystem Inter-operation



Services vs Kernels

Three sites of OS implementation

- In app's container (libraries)
- In separate containers (services)
- In central, universally shared container (kernel)

Shared nature of kernels makes them less flexible

- Single sysentry mechanism
- Inter-op requires shared abstractions
- Access controls limited

Services have natural advantages

- Filtering and refinement of operations provides finer-grained access control
- Easy to provide alternative abstractions

Example: Refining kernel privilege

Creating permanent objects in OB requires privilege

Drive letters are permanent objects (symlinks) in the ¥DosDevices directory

Q: So how does the *DefineDosDevice* API work?

A: It uses a privileged services (csrss) to create the symlink

csrss is only willing to create symlinks in ¥DosDevices

Subsystems can in general refine privileges for clients and safely share state between clients – just like kernels

No kernels: Future of OS Design?

Operating systems as a collection of libraries and services?

+ increased flexibility & extensibility

+ more robust, better failure isolation/recovery, better security

- performance of current CPUs optimized for kernels

SPACE, Pebble

– Fundamental abstractions:

	Processors,	MMUs,	trapvectors
vs.	Processes,	VM,	IPC

Back to the present…

Windows is extended primarily by adding apps and libraries (e.g. COM components)

Primary kernel extensions are for new devices and filtering existing operations

Project I explores kernel extensions

Project II explores services

Project I – writing a kernel extension

Have the Windows DDK installed for WS03 (aka WNET)

Open a new command window

set DDK=C:¥WINDDK¥37901218 (for example)

Run command: *%DDK%¥bin¥setenv %DDK% chk wnet*

In the TrivialDriver directory type: *build*

Find *trivial.sys* and *trivialapp.exe* and copy to test machine

Run *trivialapp.exe* on the test machine

You’ll see a few messages (the driver loaded/unloaded)

Do the same with TrivialDriver2

This time it waits, so start/stop *taskmgr.exe*

You will see the names of registry values that were set

Use *regedit.exe* to write some new values in HKCU

Project I - 2

Read through *Registry Callbacks.doc*

Compare TrivialDriver and TrivialDriver2

Read in the DDK documentation about the API *PsSetCreateProcessNotifyRoutine*

Have the SDK documentation handy

Modify the TrivialDriver2 driver to list the process ids of processes as they are created and exit

Then modify the app to use to print out the name of the exe for each process created (see the PSAPI functions)

This is a hit-or-miss procedure, what would be required for it to be reliable?

Anatomy of *Trivial.sys* Driver

- *DriverEntry* is called when driver is loaded
  - Creates Device object and symlink
  - Initializes a few dispatch entry points
- *TrivialCreateClose* is called for create/close IRPs
  - Since driver not stacked, only opened by name
  - Routine does nothing but process IRP correctly
- *TrivialCleanup* is also an effective no-op
- *TrivialUnload* deletes the symlink, IOMgr deletes devobj
- *TrivialDriver2* adds read and ioctl functions, and then
  - Arranges for registry callbacks
  - Maintains a buffer which can be read out

Discussion