Introducing kernel-agnostic Genode executables

Norman Feske
<norman.feske@genode-labs.com>

Outline

1. Kernel diversity - What's the appeal?

2. Bridging the gap between kernels
   - Notion of components
   - Raising the level of abstraction of IPC
   - Virtual-memory management
   - Custom tooling

3. From a uniform API to binary compatibility

4. Future prospects

Outline

1. Kernel diversity - What's the appeal?

2. Bridging the gap between kernels
   - Notion of components
   - Raising the level of abstraction of IPC
   - Virtual-memory management
   - Custom tooling

3. From a uniform API to binary compatibility

4. Future prospects

Despair-driven development

Despair-driven development

2003: Security came into focus of the L4 community
Capability-based security $\rightarrow$ new kernel generation

Genode started as the designated user land of NOVA

Despair-driven development

2003: Security came into focus of the L4 community
Capability-based security $\rightarrow$ new kernel generation

Genode started as the designated user land of NOVA

Problem: NOVA did not exist

How to build a user land for a non-existing kernel?

- Planning in terms of interim solutions
- Weak assumptions about the kernel

Despair-driven development

2003: Security came into focus of the L4 community
Capability-based security $\rightarrow$ new kernel generation

Genode started as the designated user land of NOVA

Problem: NOVA did not exist

How to build a user land for a non-existing kernel?

- Planning in terms of interim solutions
- Weak assumptions about the kernel

Approach: Target two existing kernels at once

- Opposite ends of a spectrum: Linux and L4/Fiasco
- If it works on those, it should be portable to NOVA

Reassuring experiences

Reassuring experiences

- Boosting our development
  - Quick development-test cycle on GNU/Linux
  - Debugging via GDB, strace
  - Kernel debugger on L4/Fiasco

Reassuring experiences

- Boosting our development
  - Quick development-test cycle on GNU/Linux
  - Debugging via GDB, strace
  - Kernel debugger on L4/Fiasco

- Stressing the robustness of our code
  Different kernels expose subtle problems

Reassuring experiences

- Boosting our development
  - Quick development-test cycle on GNU/Linux
  - Debugging via GDB, strace
  - Kernel debugger on L4/Fiasco

- Stressing the robustness of our code
  Different kernels expose subtle problems

- Cross-correlating bugs and performance problems

Reassuring experiences

- Boosting our development
  - Quick development-test cycle on GNU/Linux
  - Debugging via GDB, strace
  - Kernel debugger on L4/Fiasco

- Stressing the robustness of our code
  Different kernels expose subtle problems

- Cross-correlating bugs and performance problems

- Getting clarity of application-level requirements

Benefiting from a high diversity of kernels

Kernels differ in many respects:

- Hardware-platform support
- Leveraged hardware features
  Virtualization, IOMMU, SMP, TrustZone
- Performance, security, scheduling
- Implementation, License
- Community

Maintenance burden

Surprisingly little kernel-specific code!

Repository	Source lines of code
repos/	254,367
repos/base/	23,282
repos/base-fiasco/	1,563
repos/base-foc/	3,264
repos/base-linux/	3,582
repos/base-nova/	5,711
repos/base-okl4/	1,958
repos/base-pistachio/	1,869
repos/base-sel4/	3,300
repos/base-hw/	14,751

$\rightarrow$ *manageable*

Emergence of a vision

What POSIX is for monolithic OSes,
Genode may become for microkernel-based OSes.

$\rightarrow$ *Deliberate cultivation of cross-kernel interoperability*

Outline

1. Kernel diversity - What's the appeal?
2. Bridging the gap between kernels
   - Notion of components
   - Raising the level of abstraction of IPC
   - Virtual-memory management
   - Custom tooling
3. From a uniform API to binary compatibility
4. Future prospects

Overcoming prevalent assumptions

Overcoming prevalent assumptions

Application requirements are rather mysterious

- Preoccupation with scalability and performance concerns
- POSIX (?)
- Thread-local storage (?)

Overcoming prevalent assumptions

Application requirements are rather mysterious

- Preoccupation with scalability and performance concerns
- POSIX (?)
- Thread-local storage (?)

We disregarded those premises *(liberating!)*

Overcoming prevalent assumptions

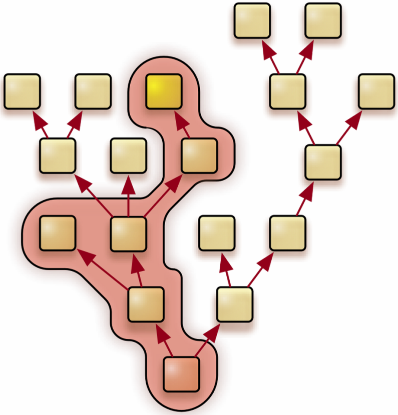
Application requirements are rather mysterious

- Preoccupation with scalability and performance concerns
- POSIX (?)
- Thread-local storage (?)

We disregarded those premises *(liberating!)*

...to be considered later.

Holistic architecture



Clean-slate design

Hiding the construction of components

Traditional: Tight user-kernel interplay

Hiding the construction of components

Traditional: Tight user-kernel interplay

Interesting at application level:

- Defining the executable to load
  $\rightarrow$ *ROM dataspace*
- Exercising control over the new protection domain
  $\rightarrow$ *Parent-child RPC interface*

Hiding the construction of components

Traditional: Tight user-kernel interplay

Interesting at application level:

- Defining the executable to load
  $\rightarrow$ *ROM dataspace*
- Exercising control over the new protection domain
  $\rightarrow$ *Parent-child RPC interface*

Approach: Satisfy those requirements, hide “loading” mechanics

Outline

1. Kernel diversity - What's the appeal?
2. Bridging the gap between kernels
   - Notion of components
   - Raising the level of abstraction of IPC
   - Virtual-memory management
   - Custom tooling
3. From a uniform API to binary compatibility
4. Future prospects

Traditional: IPC involves kernel details

Microkernel IPC ridden with technicalities and jargon

thread IDs, task IDs, portals, message registers, message tags, message dopes, message-buffer layouts, UTCBs, MTDs, hot spots, CRDs, receive windows, badges, reply capabilities, flex pages, string items, timeouts, short IPC vs. long IPC

Traditional: IPC involves kernel details

Microkernel IPC ridden with technicalities and jargon

thread IDs, task IDs, portals, message registers, message tags, message dopes, message-buffer layouts, UTCBs, MTDs, hot spots, CRDs, receive windows, badges, reply capabilities, flex pages, string items, timeouts, short IPC vs. long IPC

IDL compilers supposedly hide those details.

Traditional: IPC involves kernel details

Microkernel IPC ridden with technicalities and jargon

thread IDs, task IDs, portals, message registers, message tags, message dopes, message-buffer layouts, UTCBs, MTDs, hot spots, CRDs, receive windows, badges, reply capabilities, flex pages, string items, timeouts, short IPC vs. long IPC

IDL compilers supposedly hide those details. But they don't.

IPC from the application's perspective

IPC from the application's perspective

Genode's API level:

- Consistent and simple nomenclature
  (client, server, session, RPC object, capability)
- Synchronous RPC in the strictest sense
  (RPC stub code generated by C++ templates, no IDL)

IPC from the application's perspective

Genode's API level:

- Consistent and simple nomenclature
  (client, server, session, RPC object, capability)
- Synchronous RPC in the strictest sense
  (RPC stub code generated by C++ templates, no IDL)
- Capabilities instead of global name spaces
  (lifetime managed as C++ smart pointer)

IPC from the application's perspective

Genode's API level:

- Consistent and simple nomenclature
  (client, server, session, RPC object, capability)
- Synchronous RPC in the strictest sense
  (RPC stub code generated by C++ templates, no IDL)
- Capabilities instead of global name spaces
  (lifetime managed as C++ smart pointer)
- Asynchronous notifications without payload
  (like interrupts)

IPC from the application's perspective

Genode's API level:

- Consistent and simple nomenclature
  (client, server, session, RPC object, capability)
- Synchronous RPC in the strictest sense
  (RPC stub code generated by C++ templates, no IDL)
- Capabilities instead of global name spaces
  (lifetime managed as C++ smart pointer)
- Asynchronous notifications without payload
  (like interrupts)

→ no bit fiddling, "optimizations"

Outline

1. Kernel diversity - What's the appeal?
2. Bridging the gap between kernels
   - Notion of components
   - Raising the level of abstraction of IPC
   - Virtual-memory management
   - Custom tooling
3. From a uniform API to binary compatibility
4. Future prospects

Virtual-memory management

Traditional:

- Page-fault protocol (L4)
- Memory mappings via the kernel's IPC or map operations

Virtual-memory management

Traditional:

- Page-fault protocol (L4)
- Memory mappings via the kernel's IPC or map operations

Dataspace: Memory object referred by a capability

- Owner = creator
- Created via the root of the component tree
- Can be attached to a component's local address space
- Can be shared with others by delegating the capability
  $\rightarrow$ *shared memory*

Outline

1. Kernel diversity - What's the appeal?
2. Bridging the gap between kernels
   - Notion of components
   - Raising the level of abstraction of IPC
   - Virtual-memory management
   - Custom tooling
3. From a uniform API to binary compatibility
4. Future prospects

Using a particular kernel

Using a particular kernel

Technical aspects:

- Source distribution

Using a particular kernel

Technical aspects:

- Source distribution
- Tooling
  (configuration, build system, tool chain, custom scripts)

Using a particular kernel

Technical aspects:

- Source distribution
- Tooling
  (configuration, build system, tool chain, custom scripts)
- Kernel bindings
- Intrinsic user-level dependencies
  (ties to a particular user land)

Using a particular kernel

Technical aspects:

- Source distribution
- Tooling
  (configuration, build system, tool chain, custom scripts)
- Kernel bindings
- Intrinsic user-level dependencies
  (ties to a particular user land)
- System integration and configuration
- Booting, logging, debugging, work flows *(e. g., menu.lst)*

Using a particular kernel

Technical aspects:

- Source distribution
- Tooling
  (configuration, build system, tool chain, custom scripts)
- Kernel bindings
- Intrinsic user-level dependencies
  (ties to a particular user land)
- System integration and configuration
- Booting, logging, debugging, work flows *(e. g., menu.lst)*

$\rightarrow$ *Exploration/education costs*

Relieving the user from those technicalities

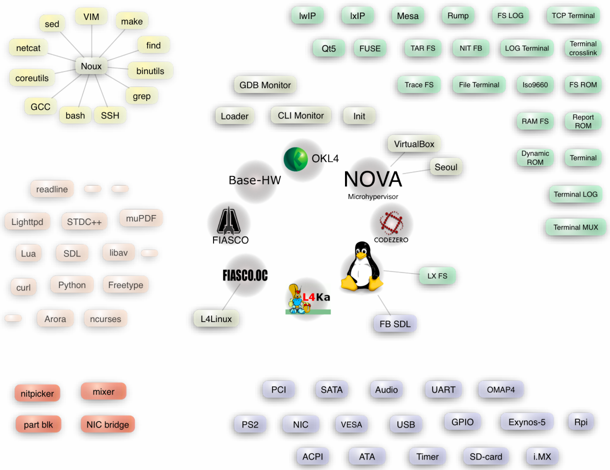
Relieving the user from those technicalities

Custom tooling

- Bullet-proof integration of 3rd-party code
  → *ports mechanism*
- Kernel-agnostic system-scenario descriptions
  → *run scripts*
- Unified tool chain
  → *blessed bare-metal C++ runtime*

The choice of the kernel is almost transparent

The choice of the kernel is almost transparent



Outline

1. Kernel diversity - What's the appeal?
2. Bridging the gap between kernels
   - Notion of components
   - Raising the level of abstraction of IPC
   - Virtual-memory management
   - Custom tooling
3. From a uniform API to binary compatibility
4. Future prospects

How the kernel taints the user land

How the kernel taints the user land

1. Inclusion of kernel headers
   - System-call bindings
   - Kernel-specific types (IDs, IPC structures, error codes)
   - Utilities

How the kernel taints the user land

1. Inclusion of kernel headers
    - System-call bindings
    - Kernel-specific types (IDs, IPC structures, error codes)
    - Utilities
2. Component code that issues system calls
    - IPC
    - Multi-threading, synchronization
    - Virtual memory management
    - Hardware access
    - Kernel-object creation/destruction

Decoupling the user land from the kernel

Decoupling the user land from the kernel

1. Clean Genode's API headers from kernel-specific artifacts
   - Uniform capability representation

Decoupling the user land from the kernel

1. Clean Genode's API headers from kernel-specific artifacts
    - Uniform capability representation
    - Generic IPC message-buffer layout

Decoupling the user land from the kernel

1. Clean Genode's API headers from kernel-specific artifacts
   - Uniform capability representation
   - Generic IPC message-buffer layout
   - Thread manipulation, synchronization

Decoupling the user land from the kernel

1. Clean Genode's API headers from kernel-specific artifacts
    - Uniform capability representation
    - Generic IPC message-buffer layout
    - Thread manipulation, synchronization
    - Hide address-space layout constraints

Decoupling the user land from the kernel

1. Clean Genode's API headers from kernel-specific artifacts
   - Uniform capability representation
   - Generic IPC message-buffer layout
   - Thread manipulation, synchronization
   - Hide address-space layout constraints

2. Galvanic separation of kernel-specific from application code
   $\rightarrow$ *distinct ELF objects*

Key element: Dynamic linker

Key element: Dynamic linker

The dynamic linker's split personality:

- Compile time: shared library
  - Linked to components
  - Satisfies dependencies on the Genode API at link time

Key element: Dynamic linker

The dynamic linker's split personality:

- Compile time: shared library
  - Linked to components
  - Satisfies dependencies on the Genode API at link time

- Runtime: static binary
  - Lives inside the component
  - Obtains and bootstraps the kernel-agnostic executable
  - Resolves references to the Genode API with itself
  - Exposes the Genode API as its library interface
  - Loads and initializes shared libraries

Key element: Dynamic linker

The dynamic linker's split personality:

- Compile time: shared library
  - Linked to components
  - Satisfies dependencies on the Genode API at link time

- Runtime: static binary
  - Lives inside the component
  - Obtains and bootstraps the kernel-agnostic executable
  - Resolves references to the Genode API with itself
  - Exposes the Genode API as its library interface
  - Loads and initializes shared libraries

free-standing Genode API $\rightarrow$ *generic ABI of the dynamic linker*

Genode's application binary interface (ABI)

ABI definition:

- Symbol names, types, and meta data
- Extracted from the concrete dynamic linker instance
- Cleaned from redundancies
  - Undefined symbols
  - Weak C++ symbols
    (template instances, inline functions, vtables, type infos)
- Cross-checked with all kernels
  - No inner-framework global symbols
  - A few kernel-specific parts remain

$\rightarrow$ *Genode ABI definition: 22 KiB*

Crossing CPU-architecture boundaries

Goal: The same ABI across all supported architectures

(x86_32, x86_64, ARM, RISC-V)

Crossing CPU-architecture boundaries

Goal: The same ABI across all supported architectures
(x86_32, x86_64, ARM, RISC-V)

Risk: Mangling of C++ symbols

Crossing CPU-architecture boundaries

Goal: The same ABI across all supported architectures *(x86_32, x86_64, ARM, RISC-V)*

Risk: Mangling of C++ symbols

Good: Almost no differences between ARM and x86_32

Crossing CPU-architecture boundaries

Goal: The same ABI across all supported architectures
(x86_32, x86_64, ARM, RISC-V)

Risk: Mangling of C++ symbols

Good: Almost no differences between ARM and x86_32

Shudder: Huge differences between x86_32 and x86_64

Life would be good without `size_t`

`size_t` = `__SIZE_TYPE__` (compiler-defined)

Life would be good without `size_t`

`size_t` = `__SIZE_TYPE__` (compiler-defined)

x86_32: `__SIZE_TYPE__` = `unsigned int`
x86_64: `__SIZE_TYPE__` = `unsigned long`

Life would be good without `size_t`

`size_t` = `__SIZE_TYPE__` (compiler-defined)

x86_32: `__SIZE_TYPE__` = `unsigned int`
x86_64: `__SIZE_TYPE__` = `unsigned long`

Mangled C++ symbols encode entire function signatures

Example: `void Connection::upgrade_ram(size_t)`

x86_32: `_ZN10Connection11upgrade_ramEj`
x86_64: `_ZN10Connection11upgrade_ramEm`

Life is (almost) good without `size_t`

No use of `__SIZE_TYPE__` by Genode API:

- `Genode::size_t` defined as `unsigned long`
  → *Genode ABI is architecture agnostic*
- Remaining problem: libc uses compiler-defined `size_t`
- Fine for C code (symbol == function name w/o arguments)
- Problem with libc-depending C++ code *(like Qt5)*

→ *Solution 1: architecture-dependent ABIs*
→ *Solution 2: tweak the compiler*

Generalization of the ABI mechanism

Generalization of the ABI mechanism

Build system support:

- ABI definition is translated to an assembly file
  (almost architecture independent)

Generalization of the ABI mechanism

Build system support:

- ABI definition is translated to an assembly file
  (almost architecture independent)
- Assembly file is compiled/linked into an `.abi.so` file
  (shared library that contains only symbols but no code)

Generalization of the ABI mechanism

Build system support:

- ABI definition is translated to an assembly file *(almost architecture independent)*
- Assembly file is compiled/linked into an `.abi.so` file *(shared library that contains only symbols but no code)*
- Library-using targets are linked against the `.abi.so` file instead of the real library

Generalization of the ABI mechanism

Build system support:

- ABI definition is translated to an assembly file
  (almost architecture independent)
- Assembly file is compiled/linked into an `.abi.so` file
  (shared library that contains only symbols but no code)
- Library-using targets are linked against the `.abi.so` file instead of the real library
- ABI formalism for arbitrary libraries!
  (merely add an ABI definition for a library)

Generalization of the ABI mechanism

Build system support:

- ABI definition is translated to an assembly file
  (almost architecture independent)
- Assembly file is compiled/linked into an `.abi.so` file
  (shared library that contains only symbols but no code)
- Library-using targets are linked against the `.abi.so` file instead of the real library
- ABI formalism for arbitrary libraries!
  (merely add an ABI definition for a library)

→ *Targets can be built without the libraries they depend on.*

Immediate benefits

Build directory used to depend on kernel and hardware platform.

Immediate benefits

Build directory used to depend on kernel and hardware platform.

New unified build directories:

- Depend only on hardware platform

Immediate benefits

Build directory used to depend on kernel and hardware platform.

New unified build directories:

- Depend only on hardware platform
- Kernel-agnostic targets are linked dynamically
  (almost all components)
- Kernel-specific targets are named after the kernel
  (ld-nova.lib.so, core-nova, timer driver)
  $\rightarrow$ *build results can peacefully coexist*

Immediate benefits

Build directory used to depend on kernel and hardware platform.

New unified build directories:

- Depend only on hardware platform
- Kernel-agnostic targets are linked dynamically
  (almost all components)
- Kernel-specific targets are named after the kernel
  (ld-nova.lib.so, core-nova, timer driver)
  $\rightarrow$ *build results can peacefully coexist*
- Choice of kernel not before running a scenario:
  `make run/demo KERNEL=nova`

Outline

1. Kernel diversity - What's the appeal?
2. Bridging the gap between kernels
   - Notion of components
   - Raising the level of abstraction of IPC
   - Virtual-memory management
   - Custom tooling
3. From a uniform API to binary compatibility
4. Future prospects

Future prospects

Package management

- Distinction between source and API/ABI packages
  $\rightarrow$ *Loose coupling of packages*
- Binary packages independent of the used kernel

Future prospects

Package management

- Distinction between source and API/ABI packages
  $\rightarrow$ *Loose coupling of packages*
- Binary packages independent of the used kernel

Multiple levels of API/ABI stability

Future prospects

Package management

- Distinction between source and API/ABI packages
  $\rightarrow$ *Loose coupling of packages*
- Binary packages independent of the used kernel

Multiple levels of API/ABI stability

Two orthogonal directions

1. Successive hardening of the foundation, transparent to users
2. Scaling the software stack with a fixed target

Unique solutions, enabled by Free Software

Unique solutions, enabled by Free Software

Shaping the entire vertical software stack:

- Tool chain $\leftrightarrow$ Work-flow automation $\leftrightarrow$ Quality assurance
- Build system $\leftrightarrow$ Source-code management $\leftrightarrow$ Package management
- Dynamic linker *(cross-kernel binary compatibility)*
- C runtime, C++ runtime *(encapsulating legacies)*
- VFS infrastructure *(component-level customiztions)*
- Init and system configuration *(session routing)*
- Genode ABI and API *(enforcing a safe C++ dialect)*
- Kernel *(base-hw, scheduling, kernel-resource management)*
- Component interfaces *(multi-component applications)*
- User interface $\leftrightarrow$ System management

Unique solutions, enabled by Free Software

Shaping the entire vertical software stack:

- Tool chain $\leftrightarrow$ Work-flow automation $\leftrightarrow$ Quality assurance
- Build system $\leftrightarrow$ Source-code management $\leftrightarrow$ Package management
- Dynamic linker *(cross-kernel binary compatibility)*
- C runtime, C++ runtime *(encapsulating legacies)*
- VFS infrastructure *(component-level customiztions)*
- Init and system configuration *(session routing)*
- Genode ABI and API *(enforcing a safe C++ dialect)*
- Kernel *(base-hw, scheduling, kernel-resource management)*
- Component interfaces *(multi-component applications)*
- User interface $\leftrightarrow$ System management

$\rightarrow$ Cross-pollination between different levels

Unique solutions, enabled by Free Software

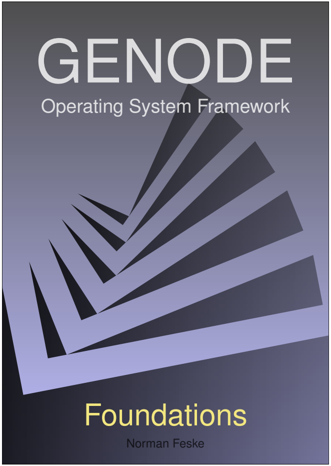
Shaping the entire vertical software stack:

- Tool chain $\leftrightarrow$ Work-flow automation $\leftrightarrow$ Quality assurance
- Build system $\leftrightarrow$ Source-code management $\leftrightarrow$ Package management
- Dynamic linker *(cross-kernel binary compatibility)*
- C runtime, C++ runtime *(encapsulating legacies)*
- VFS infrastructure *(component-level customiztions)*
- Init and system configuration *(session routing)*
- Genode ABI and API *(enforcing a safe C++ dialect)*
- Kernel *(base-hw, scheduling, kernel-resource management)*
- Component interfaces *(multi-component applications)*
- User interface $\leftrightarrow$ System management

$\rightarrow$ Cross-pollination between different levels

$\rightarrow$ **Simple and holistic solutions!**

The Book “Genode Foundations”



https://genode.org/documentation/genode-foundations-16-05.pdf

Thank you

Genode OS Framework

https://genode.org

Genode Labs GmbH

https://www.genode-labs.com

Source code at GitHub

https://github.com/genodelabs/genode