Genode OS Framework - Architecture and Components

David Werner

January 7, 2020

Technische Universität München

Outline

Copyright Notice (1)

Basics

Basics - Foundation

Contradicting properties that are desirable for OSs:

- Assurance $\longleftrightarrow$ Scalability
- Security $\longleftrightarrow$ Ease of use
- Utilization $\longleftrightarrow$ Accountability

The Genode architecture resolves these contradictions.

Basics - Key techniques

Genode architecture combines multiple techniques:

- Usage of microkernels
- Capability-based security
- Kernelization
- Virtualization
- Management of budgets
- Trading and tracking of resources
- Application-specific trusted computing base

Basics - Features

The Genode OS Framework is the implementation of the Genode Architecture:

- Scales from small embedded systems to general-purpose systems
- Collection of building blocks which enable the composition of sophisticated systems
- Support for multiple CPU architectures (x86, ARM, RISC-V)
- Can be deployed on most L4-microkernels
- Supports virtualization

Basics - Source Tree Structure

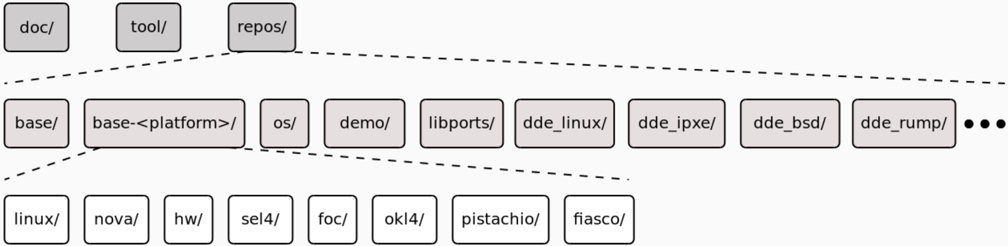

Fig. 1: [genode]

Architecture

Architecture - Remote Procedure Call objects

Genode components exist inside *protection domains* (PDs)

- Isolated execution environment
- *RPC objects* can be created inside PDs
- Each of them provides interface to access it
- Access from outside of the PD/component possible

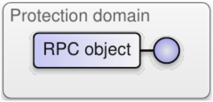

Fig. 2: [genode]

Architecture - Capability-based security

Genode uses capabilities as security mechanism:

- Own capability model (independent of the kernel)
- Capabilities reference *RPC objects*
- The possession of a capability allows the invocation of the corresponding RPC object
- Components store their capabilities in capability maps

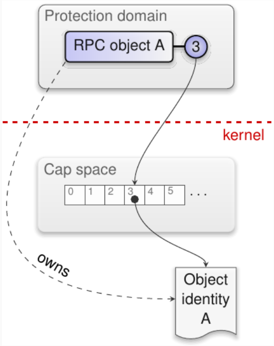

Fig. 3: [genode]

Architecture - Capability delegation

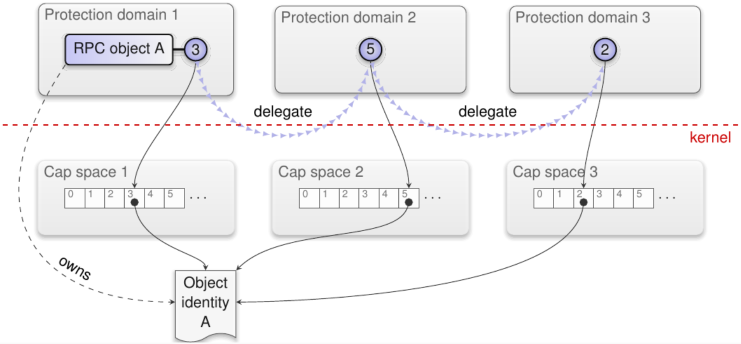

Fig. 4: [genode]

Architecture - Capability invocation

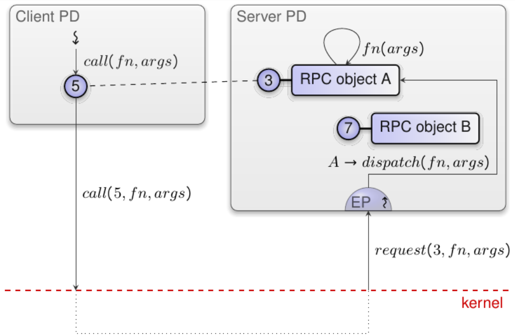

Fig. 5: [genode]

Architecture - Recursive tree structure

All components within a Genode-based system are organized in a tree structure:

- Parent-Child relationship between components
- This relationship has two aspects:
  - Responsibility
  - Control
- Parents manage child's service usage (sessions)
- *Core* and *Init* form the root of the component tree

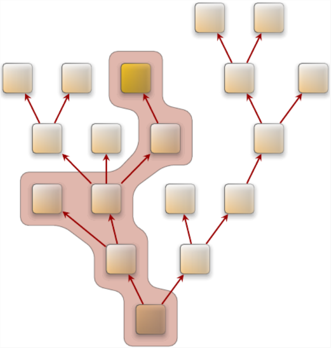

Fig. 6: [genode]

Architecture - Resource trading

Genode does not abstract from physical resources:

- Access to resources is arbitrated by the use of budgets
- E.g. each component has a memory budget which determines the amount of memory the component is allowed to allocate
- Budgets are stored in the PD

Resource trading:

- Child components need to receive their budgets from their parents
- For some services the client has to delegate some of its budgets (e.g. memory budget) to the server

Architecture - Core

Core is the first user-level component:

- Directly created by the microkernel
- Root of the component tree
- Has access to raw physical resources
- Core exposes these resources to the system as services
- Therefore *Core* is a server
- But: Almost free of policy (Similar to the underlying microkernel)

Architecture - Dataspaces

A *Dataspace (DS)* is a *RPC object* in core:

- Represents a contiguous physical address-space region
- Base address and size are subjected to the granularity of physical pages (typically 4 KiB)
- Created and managed via core services
- Each component in possession of the *DS* capability can make the *DS* content visible in its local address space

Architecture - Region maps

A *Region map*:

- RPC object within core
- Represents the layout of a virtual address space
- The size is defined at creation time
- Implicitly created as part of the protection domain (for usage as local address space)
- Can also be created explicitly (see slide on region-map management)

Architecture - Access to boot modules

During the initial machine bootstrap the bootloader loads all boot modules (binaries) into the memory:

- To make boot modules available, *Core* provides the *ROM service*
- Each *ROM session* gives access to a single boot module (ROM module)
- The module name is handed as parameter on session creation
- The session allows to retrieve a dataspace which contains the binary

Architecture - Protection domains

A *protection domain* (*PD*) corresponds to a unit of protection within the Genode system:

- Core offers *PD service* to create *PDs*
- Typically 1-to-1 relationship between a component and a *PD session*
- The session consists of:
  - Virtual memory address space (in form of a region map)
  - The capability space
  - Budgets (Memory and Capability)

Architecture - Region-map management

The *RM service* of *Core*:

- Allows to create additional region maps
- These region maps are also referred to as *managed dataspaces*
- Managed DSs can be attached to the virtual address space (just like 'normal' DSs) but are not backed by physical address
- Generalization of nested page tables

Architecture - Processing-time allocation

To enable the allocation of processing time, *Core* provides the *CPU service*:

- A *CPU session* allows for the creation, manipulation and destruction of threads
- At construction time of the session the CPU affinity can be determined
- Created threads are represented by thread capabilities

Architecture - Access to device resources

Core offers three services for the realization of user-level device drivers:

- *IO_MEM*
  - A *IO_MEM session* provides a dataspace to memory-mapped I/O and BIOS regions
  - Each memory range is handed out only once
- *IO_PORT*
  - The service enables fine-grained assignment of ports to components
  - Each session corresponds to the exclusive access right to a port
- *IRQ*
  - Each *IRQ session* corresponds to an interrupt (1-to-1 relationship)
  - Provides means to wait for interrupts

Architecture - Inter-process communication

Genode provides three mechanisms for Inter-process communication (IPC):

- Synchronous remote procedure calls (RPC)
- Asynchronous notifications
- Shared memory

Architecture - Remote procedure calls (1)

Genode's RPC mechanism:

- Layered structure
  - Base is the kernel's IPC mechanism
  - Each layer above adds Genode semantics
  - Whole stack of layers provides the full Genode IPC mechanism
- Notion of typed capabilities

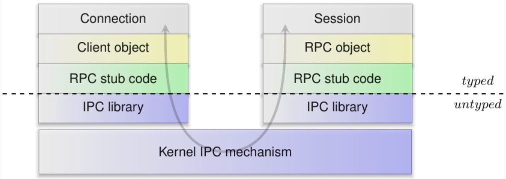

Fig. 7: [genode]

RPC object creation:

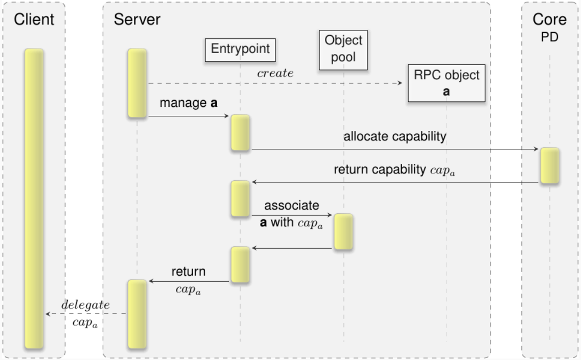

Fig. 8: [genode]

Architecture - Remote procedure calls (3)

RPC object invocation:

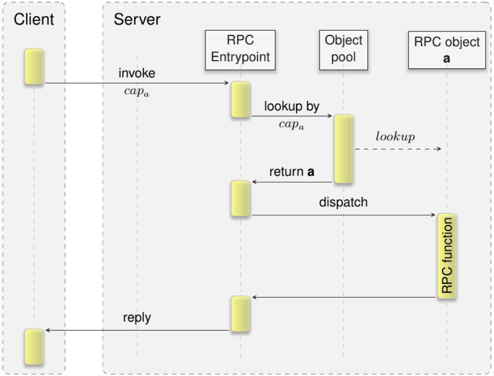

Fig. 9: [genode]

Architecture - Asynchronous notifications (1)

Asynchronous notifications (Signals):

- Enable waiting for multiple conditions
- Provide means to signal events to untrusted parties
- Signals carry no payload (unlike RPCs)
- Signaling is realized by *Signal handlers* which create *Signal contexts*
- *Signal context* capabilities can be delegated which enables the receivers to send notifications to the *handler* (the process is depicted on the next slide)

Architecture - Asynchronous notifications (2)

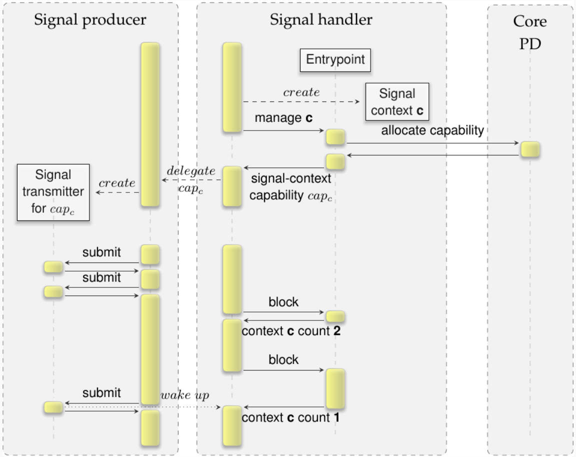

Fig. 10: [genode]

Architecture - Shared Memory (1)

Shared Memory:

- Enables components to propagate large amounts of data across component boundaries
- No active involvement of the kernel
- Realized by sharing a common dataspace
  - Allocated by the server
  - Memory budget from child is used

Architecture - Shared Memory (2)

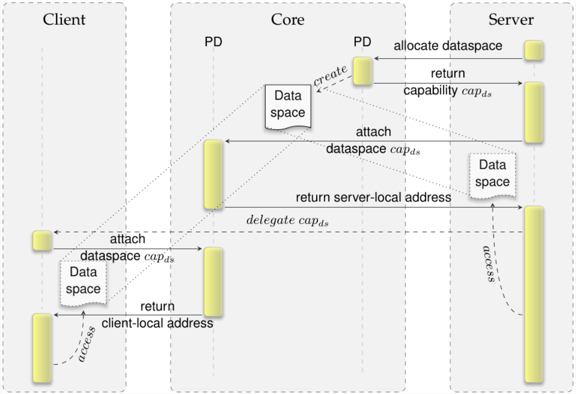

Fig. 11: [genode]

Architecture - Combination of base mechanisms

Combinations of the three IPC mechanisms are possible:

- Asynchronous state propagation
- Synchronous bulk transfer
- Asynchronous bulk transfer (packet streams)

Components

Components - Basics (1)

A component in Genode

- Composed by using Genode’s architecture
- Represents a sophisticated building block of the system:
  - Operating system functionality
  - Applications
- Resides in a dedicated protection domain
- Interacts with other components

Components - Basics (2)

The functional scope of a component depends on several factors:

- Security
- Performance
- Reusability

Versatility of component-based systems comes from:

- Granularity of componentization (determined by developer)
- Composability of components
- But: Granularity and Composability need to be designed in a smart way as small component interfaces are desirable

Components - Basics (3)

Basic parts of a Genode component:

- *PD session* representing the component's protection domain
- *ROM session* with the executable binary
- *CPU session* for creating the initial thread of the component

These sessions are obtained by the parent of the new (child) component.

Components - Creation (1)

Parent components want to create new child component - Initial situation:

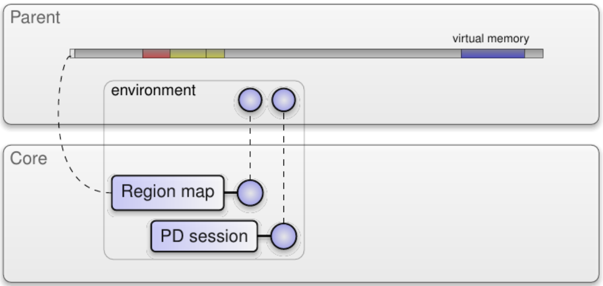

Fig. 12: [genode]

Components - Creation (2)

First step: Parent obtains the component executable binary

- *ROM session* is created
- The child's executable binary is retrieved as dataspace

Second step: Creation of the child's designated *PD session*

- Fresh *PD session* is created by the parent
- Transfer of memory budget
- Transfer of capability budget

Components - Creation (3)

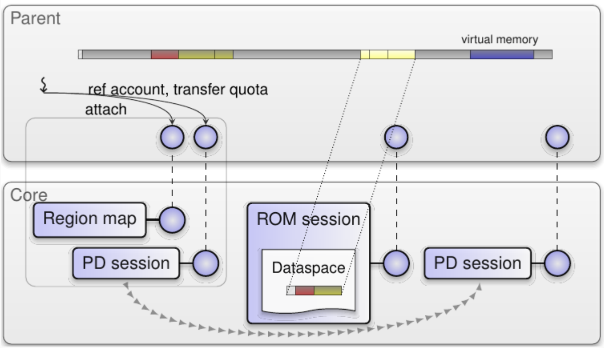

Fig. 13: [genode]

Components - Creation (4)

Third step: Constructing the child's address space

- Attaching read-only segments (program code from the child's ROM dataspace)
- Attaching read-writable segments
  - Obtained from ELF binary (ROM dataspace)
  - Allocated with the child's memory budget

Components - Creation (5)

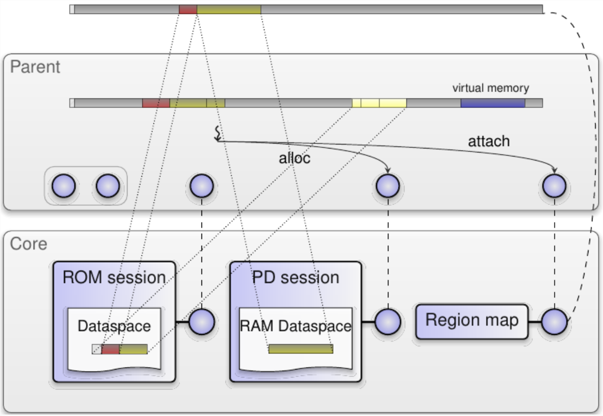

Fig. 14: [genode]

Components - Creation (6)

Fourth step: Creating the initial thread

- The child's *CPU session* is created by the parent
- Afterwards the initial thread is created
- Immediately after its creation the thread remains inactive until it is configured
- The parent installs a so-called parent capability in the child to establish a communication channel
- The fourth step is finished by starting the child's initial thread

Components - Creation (7)

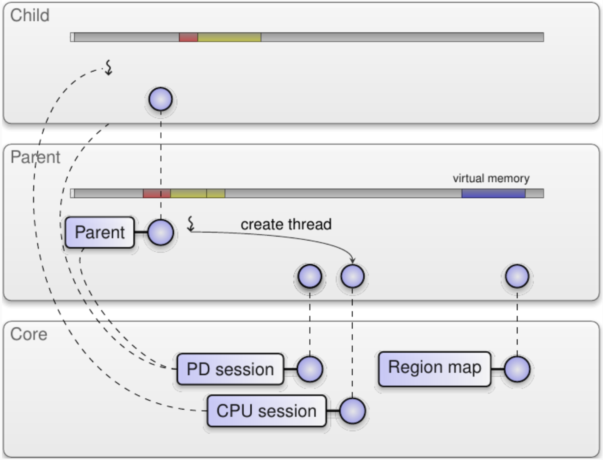

Fig. 15: [genode]

Parent-Child Relationship

Parent-Child Relationship - Revision

We already took a look at the component organization in Genode:

- Hierarchical structure
- Parent-Child relationship $\Rightarrow$ Ownership
- This relationship is two-fold:
  - Responsibility
  - Control

Parent-Child Relationship - Component ownership

Every component is owned by another one:

- Except for root (Core)
- Components can own multiple children
- Children have to inherently trust their parents

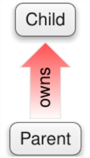

Fig. 16: [genode]

Parent-Child Relationship - Responsibility (1)

Responsibility for the child:

- Each component requires:
  - Physical resources
  - Kernel data structures
- Parent has to provide budgets for these resources
  - Child budgets are created by dividing own budgets
  - It is the parent's task to balance the budgets

Parent-Child Relationship - Responsibility (2)

- Parent defines aspects of child’s execution
  - E.g. Affinity
- Parent is the primary point of contact for the child
  - The child’s session requests are routed to the parent
  - The parent then needs to make sure that the service is provided to the child

Parent-Child Relationship - Control

Control over the child:

- Child is created out of its parent's resources
- Parent may destroy the child at any time
- Parent controls the relationships of its children to other components
- A session request of a child can be denied by the parent

Parent-Child Relationship - Parent interface (1)

Interface to the parent component:

- The parent capability is the only mean for communication after the creation of a child
- Other components are unknown to the child

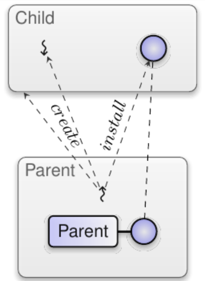

Fig. 17: [genode]

Parent-Child Relationship - Parent interface (2)

- Parents are part of the trusted computing base of their children
- Parents do not need to trust their children
- From the child perspective, the parent is as powerful as the kernel

Parent-Child Relationship - Resource trading

In Genode, budgets are used to delegate authority over physical resources:

- Parent components use their budgets to create their children
- Therefore: Mechanism to delegate budgets from parents to children required
- *PD sessions* are used to account for the budgets
- *PD sessions* provide RPC calls to move portions of budgets between them

Services and Sessions

Services and Sessions - Revision

We already talked about services and sessions:

- Services are functionality that server components provide to client components
- Sessions are instances of services
- Servers need to announce their service in order to make it accessible
- Clients acquire session capabilities via issuing a session request to their parent

Services and Sessions - Root interface

In order to provide services, server components need to provide a *root interface*:

- Offers functions to create and destroy sessions (session RPC objects)
- The root interface is implemented by a RPC object (root component)
- Announcing the service includes the delegation of the respective capability to the parent

Services and Sessions - Service announcement

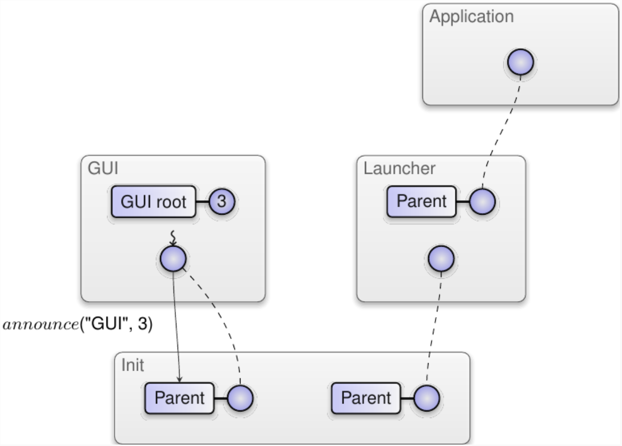

Fig. 18: [genode]

Services and Sessions - Session request

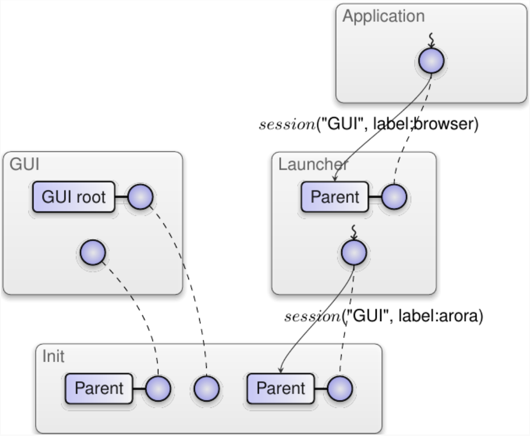

Fig. 19: [genode]

Services and Sessions - Session creation

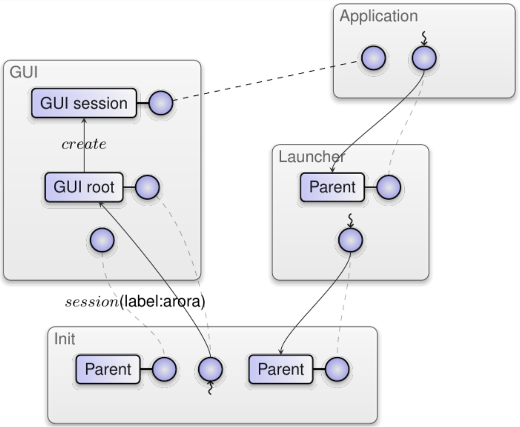

Fig. 20: [genode]

Server-Client Relationship

Server-Client Relationship - Trust (Client)

In role of a client exist different trust aspects than in the role of a child:

- No awareness of the real identity of a server
- Unable to judge whether a server is trustworthy or not
- But: Session capability was obtained via parent $\Rightarrow$ Its integrity is granted
- The client is able to decide which information is provided to a server via IPC
- Remaining problem: Flow of execution is handed to servers

Server-Client Relationship - Trust (Server)

Servers do generally <u>not</u> trust their clients:

- Client components should be expected to misbehave for two reasons:
  - Servers need to validate RPC arguments
  - Servers should never make themselves dependent on clients
  - E.g. invocation of received capabilities
- Advantage of server components: Receivers of RPC calls
- $\Rightarrow$ No blocking (and waiting)

Server-Client Relationship - Ownership of sessions

Session ownership:

- All session related RPC objects are owned by the server:
  - Session RPC object itself
  - RPC objects created by the session
- Clients cannot dictate the closing of a session
- Session closing procedure is used (similar to session creation)
- Common parent serves as broker

Server-Client Relationship - Resource trading (1)

Resource trading also exists between clients and servers:

- Goal: Resilience against client-driven resource-exhaustion attacks
- Realized by using *session quotas*:
  - Clients can attach a portion of their memory quota to session requests
  - Server uses this quota to allocate RPC objects regarding the session
  - Therefore: Client pays the memory for its session
  - Session quota can be upgraded

Device Drivers

Device Drivers - Basics (1)

Device Drivers in Genode:

- Translate device interfaces to Genode interfaces
- Typically comprise:
  - Driving of the device's state machine
  - Notification of device related events
  - Means to transfer data from and to the device
- Usually low complexity
- No hardware multiplexing / Single client per driver component

Device Drivers - Basics (2)

Drivers access the device via core services:

- IO_MEM
- IO_PORT
- IRQ

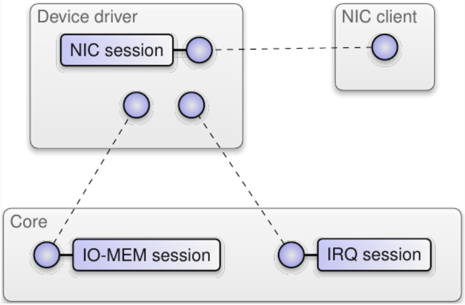

Fig. 21: [genode]

Device Drivers - Platform Driver

Purpose of the Platform Driver:

- Solves three problems which exceed the scope of a normal driver due to affecting the whole system:
  - Device enumeration
  - Discovery of interrupt routing
  - Initial hardware setup
- It provides:
  - An interface to the PCI bus
  - An IRQ service that transparently applies interrupt routines
  - Means to allocate DMA buffers

Device Drivers - Interrupt handling

Most device drivers need to respond to sporadic events:

- Produced by the device
- Propagated to the CPU

The interrupt is obtained via core’s IRQ service:

- No direct usage
- IRQ service of the platform driver

Device Drivers - Direct memory access (1)

Direct Memory Access (DMA):

- Used to transfer large amounts of data from devices to memory
- CPU does not actively participate (no copying)
- MMU is not involved
- Optimizes throughput of the system bus by using burst transfers
- Can be used to establish direct data paths between devices
- Risk: Corruption of the physical memory due to misguided DMA

DMA with memory corruption:

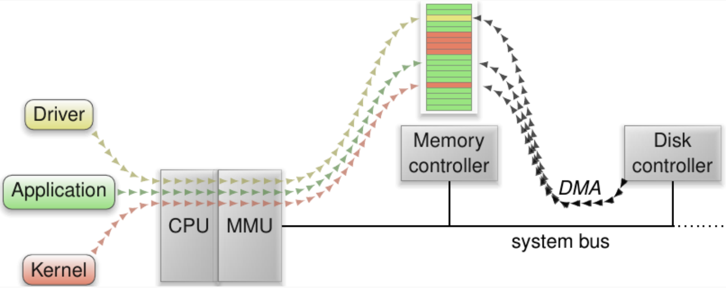

Fig. 22: [genode]

Device Drivers - DMA in component-based systems (1)

DMA in a component-based OS seems inconsistent with its principles:

- Each system component is encapsuled within a dedicated user-level address space
- If a component fails, other components are unaffected
- But: DMA is a loophole
- Especially drivers (sources of most OS bugs) are using DMA

Component-based systems are still reasonable:

- Bugs unrelated to DMA are still confined in the driver component
- Isolation of drivers from other OS parts still reduces the attack surface
- Modern hardware incorporates IOMMUs

DMA with an IOMMU:

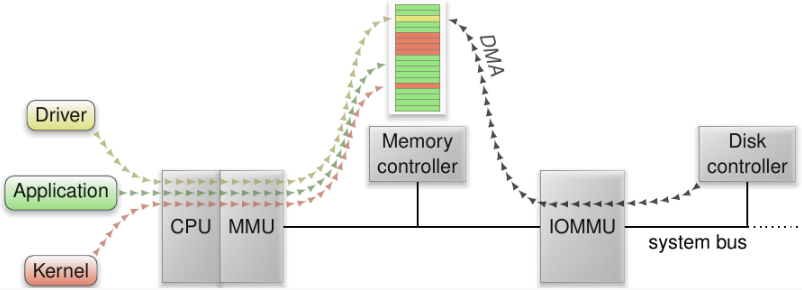

Fig. 23: [genode]

Device Drivers - Usage Example

Terminal provides the translation

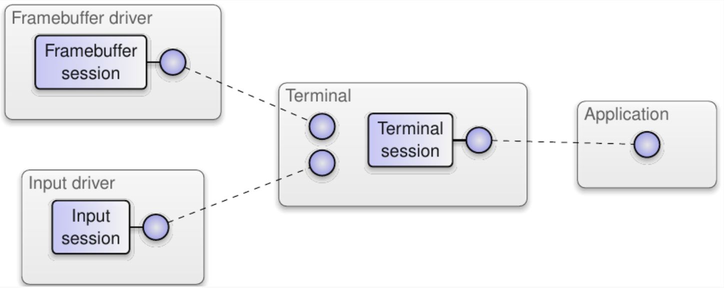

Fig. 24: [genode]

Further Reading

Further Reading - Genode OS Framework

Genode website:
`https://genode.org/`

Genode foundations:
`https://genode.org/documentation/architecture/index`