Memory, IPC, and L4Re

Björn Döbel
What we talked about so far

- Microkernels
- Fiasco.OC
- Threads in Fiasco.OC
Lecture Outline

- Some more Fiasco.OC fundamentals
  - Inter-Process Communication
  - Memory Management

- L4Re Mechanisms to implement clients and servers
• Threads are the basic building blocks of Fiasco.OC/L4Re applications.

• Threads need to communicate to achieve their goals:
  – Data exchange
  – Synchronization
  – Sleep
  – Handle hardware/software interrupts
  – Grant resource access

• This is all done using the **Inter-Process Communication (IPC)** mechanisms provided by Fiasco.OC.

• Jochen Liedtke: “**IPC performance is the master.**”
IPC: Issues

• How to implement it?

• How to integrate with languages / environments?

• How to manage IPC securely?

• How to make it fast?
Asynchronous IPC
- Example: Mach
- Pro:
  - fire&forget messaging
- Cons:
  - 2 copies: into kernel and out of kernel
  - DoS attack possible if kernel does not manage IPC properly

Synchronous IPC
- Example: L4
- Pros
  - No double copy
  - No kernel memory involved
- Con
  - Partners need to synchronize
**Basics**

- **S** → **R**: send
- **S** ← **R**: receive from *(closed wait)*
- **?** ← **R**: receive any *(open wait)*

**Special cases for client/server IPC**

- **call** := send + recv from
- **reply and wait** := send + recv any
IPC in Fiasco.OC

- IPC building block: kernel **IPC Gate** object
- Creation:
  ```cpp
  L4::Factory* factory = L4Re::Env::env()->factory();
  l4_msgtag_t tag = factory->create_gate(
      target_cap, thread_cap, label);
  ```
IPC in Fiasco.OC

- IPC building block: kernel **IPC Gate** object
- Creation:

  ```
  L4::Factory* factory = L4Re::Env::env()->factory();
  l4_msgtag_t tag = factory->create_gate(
      target_cap, thread_cap, label);
  ```

The factory is a kernel object to create new kernel objects.
IPC in Fiasco.OC

- IPC building block: kernel **IPC Gate** object
- Creation:
  
  ```c
  L4::Factory* factory = L4Re::Env::env()->factory();
  l4_msgtag_t tag = factory->create_gate(
      target_cap, thread_cap, label);
  ```

L4Re::Env::env() gives a program access to its initial capability environment.
IPC in Fiasco.OC

- **IPC building block**: kernel **IPC Gate** object
- **Creation**:
  ```cpp
  L4::Factory* factory = L4Re::Env::env()->factory();
  l4_msgtag_t tag = factory->create_gate(target_cap, thread_cap, label);
  ```

A thread may be attached to multiple IPC gates. To identify through which gate a message arrived, every gate may get a dedicated “label” assigned.
• IPC building block: kernel **IPC Gate** object

• Creation:

```cpp
L4::Factory* factory = L4Re::Env::env()->factory();
l4_msgtag_t tag = factory->create_gate(
    target_cap, thread_cap, label);
```

A free slot in the task's capability table. The newly created object is referenced by this slot. Usually allocated using:

```cpp
L4Re::Util::cap_alloc.alloc()
```
IPC in Fiasco.OC

• IPC building block: kernel **IPC Gate** object

• Creation:
  ```cpp
  L4::Factory* factory = L4Re::Env::env()->factory();
  l4_msgtag_t tag = factory->create_gate(
      target_cap, thread_cap, label);
  ```

• Receiving a message:
  - Open wait using thread's UTCB
  - Unblocks upon message arrival → use label to distinguish IPC gates

• Sending a reply:
  - Normally, one would need a capability to send the reply to. But we only have the (one-way) IPC gate!
  - Fiasco.OC stores last caller's reply capability internally.
  - May be used until next wait() starts.
• **User-level Thread Control Block**
• Set of “virtual” registers
• Message Registers
  – System call parameters
  – IPC: direct copy to receiver
• Buffer registers
  – Receive flexpage descriptors
• Thread Control Registers
  – Thread-private data
  – Preserved, not copied
IPC: The Message Tag

- Special descriptor that defines which parts of the UTCB are valid upon an IPC call.
- Protocol: user-defined “message type”
- Words: number of valid items in message registers. Directly copied to receiver.
- Items: number of valid items in buffer registers. Handled specially by the kernel.
• **Timeouts**
  – Maximum amount of time to wait for partner to become ready.
  – Most common: 0 and NEVER
  – Separate timeouts for send and receive phases

• **Exceptions**
  – Fiasco.OC can notify an exception handler thread of special events that happened to another thread
    • Memory management: page faults
    • Virtualization exceptions
Happily writing IPC code...

```c
/* Arguments: 1 integer parameter, 1 char array with size */
int FOO_OP1_call(l4_cap_idx_t dest, int arg1, char *arg2, unsigned size) {
    int idx = 0; // index into message registers

    // opcode and first arg go into first 2 registers
    l4_utcb_mr()->mr[idx++] = OP1_opcode;
    l4_utcb_mr()->mr[idx++] = arg1;

    // tricky: memcpy buffer into registers, adapt idx according
    // to size (XXX NO BOUNDS CHECK!!!)
    memcpy(&l4_utcb_mr()->mr[idx], arg2, size);
    idx += round_up(size / sizeof(int));

    // create message tag (prototype, <idx> words, no bufs, no flags)
    l4_msgtag_t tag = l4_msg_tag(PROTO_FOO, idx, 0, 0);
    return l4_error(l4_ipc_call(dest, l4_utcb(), tag, TIMEOUT_NEVER));
}
```
Happily ... ?

- This is dull and error-prone work!
- It can be automated:
  - Take 1: Interface Definition Language (IDL)
    ```
    interface FOO {
        int OP1(int arg1,
                [size_is(arg2_size)] char *arg2,
                unsigned arg2_size);
    }
    ```
  - IDL Compiler automatically generates IPC code.
    - Worked for older versions of L4(Env):
      Dice – **DROPS IDL Compiler**
    - Currently out of service. :(
Integrating IPC with C++

- L4Re (and Genode!) use C++ stream operators to make IPC fancier.
  - Compiler takes care of serializing basic data types into the right place.

- Stream library can abstract from the underlying kernel
  - reuse the same IPC code on different kernels
  - heavily used in Genode

- Users work with special IPC Stream objects.

- Good news: This can even be combined with an IDL compiler.
```c
int Foo::op1(l4_cap_idx_t dest, int arg1,
            char *arg2, unsigned arg2_size)
{
    int result = -1;
    L4_ipc_iostream i(l4_utcb()); // create stream
    i << Foo::Op1 // args → buffer
        << arg1
        << Buffer(arg2, arg2_size);

    // perform call
    int err = l4_error(i.call(dest));
    if (!err)
        i >> result;
    return result;
}
```
```cpp
int Foo::dispatch(L4_ipc_iostream& str, l4_msgtag_t tag) {
    // check for invalid invocations
    if (tag.label() != PROTO_FOO)
        return L4_ENOSYS;

    int opcode, arg1, retval;
    Buffer argbuf(MAX_BUF_SIZE);

    str >> opcode;
    switch(opcode) {
        case Foo::Op1:
            str >> arg1 >> argbuf;
            // do something clever, calculate retval
            str << retval;
            return L4_EOK;
        // .. more cases ..
    }
}
```
L4Re: IPC Framework

- L4Re provides IPC framework
  - Implementation (& docs):
    l4/pkg/cxx/lib/ipc/include

  - C++ streams → ipc_stream
    - Marshalling/unmarshalling of basic data types
      (C++ POD, C strings)

  - C++ server framework → ipc_server

- Usage example: l4/pkg/examples/clntsrv
L4Re: Server Infrastructure

• C++ classes to implement servers
  – Basic message reception
    • Optional advanced error handling etc.
  – Handling of server objects
    • Varying types
    • Varying amount of objects
• Implements a basic server loop
  \[
  \textbf{while} \ 1 \ { \\
  \hspace{1cm} m = \text{recv\_message}(); \\\n  \hspace{1cm} \text{ret} = \text{dispatch}(m.\text{tag}(), \\text{ios}); \\\n  \hspace{1cm} \text{reply}(m, \ \text{ret}); \\
  \}
  \]

• Users need to provide a \texttt{dispatch()} function.
  – If curious, see \texttt{L4::Ipc\_svr::Direct\_dispatch}

• Plus: Loop hooks
  – Allows you to specify callbacks for
    \begin{itemize}
    \item IPC error
    \item Timeout
    \item Prepare to wait
    \end{itemize}
  – You should get away with \texttt{L4::Default\_look\_hooks}
Server objects

- Servers may provide
  - Different services
  - To different clients
  - Through different server objects

- A server object is basically a wrapper for a capability:
  \( \text{L4::Server\_object::obj\_cap()} \)

- Server object implementations may vary
  - Need a dedicated \text{dispatch()} function per object

- But how does the server decide at runtime which function to call?
Server objects are stored in per-server object registry

Registry provides a function to find an object by an ID

```cpp
L4::Basic_registry::Value
L4::Basic_registry::find(l4_umword_t id);
```

ID is the label of the IPC gate through which a message arrived.

Basic_registry is **basic**:
- No ID management
- ID is simply interpreted as a pointer to a local object
Advanced object management

- Advanced servers want to bind objects to
  - IPC gates or
  - Names specified in the Lua init script

- Use object pointer as label for IPC gate

```cpp
L4::Cap<void> register_obj(L4::Server_object *o, char const *service);

L4::Cap<void> register_obj(L4::Server_object *o);

bool unregister_obj(L4::Server_obj *o);
```
There's a utility class representing a server with an object registry:

```
L4Re::Util::Registry_server
```
Server Implementation for Beginners

- Declare a server:

  ```cpp
  static L4Re::Util::Registry_server<> server;
  ```

- Implement a session factory object:

  ```cpp
  class MySessionServer
  : public L4::Server_object
  {
  ..
  };
  ```

- Implement a dispatch function:

  ```cpp
  int MySessionServer::dispatch(l4_umword_t o,
      L4::Ipc_iostream& ios)
  {
  ..
  };
  ```
Connecting Client & Server

- L4Re setups are usually started using an init script → Lua programming language.

- So far, we put a single binary as direct parameter to moe:

  roottask moe --init=rom/hello

→ instead of --init, we can pass a Lua script:

  roottask moe rom/hello.lua

  ...

  module hello.lua
Initialization: Lua

• Simple script to start hello:

```lua
require("L4");
L4.default_loader:start( {}, "rom/hello" );
```

Import the L4 Lua library
Simple script to start hello:

```lua
require("L4");
L4.default_loader:start( {}, "rom/hello" );
```

The `default_loader` provides functionality to launch ELF executables.
Simple script to start hello:

```lua
require("L4");
L4.default_loader:start( {}, "rom/hello" );
```

The `start()` function takes 2 parameters:

1. An environment for the application. (More later.)
2. The file name of the binary to launch.
• Lua interpreter establishes communication and passes them through the environment.

```lua
local ld = L4.default_loader;
local the_server = ld:new_channel();
ld:start( { caps = { calc_server = the_server:svr() },
            log = { "server", "blue" },
            "rom/server" } );
ld:start( { caps = { calc_server = the_server },
            log = { "client", "green" },
            "rom/client" } );
```

`new_channel()` creates an IPC gate. No thread is bound to it yet.
L4/Lua: Connecting Services

caps = {} defines additional capabilities that are passed to the application on startup. The syntax is <name> = <object>, where name is later used to query this capability at runtime and object is a reference to an object.

For IPC channels, 'object' is the sender side of the channel, while 'object:svr()' refers to the server/receiver side.

```lua
local ld = L4.default_loader;
local the_server = ld:new_channel();
ld:start( {
caps = { calc_server = the_server:svr() },
log = { "server", "blue" } },
"rom/server" );

ld:start( { caps = { calc_server = the_server },
log = { "client", "green" } },
"rom/client" );
```
L4/Lua: Connecting Services

- Lua interpreter establishes communication channels and passes them through the environment.

```lua
local ld = L4.default_loader;
local the_server = ld:new_channel();
ld:start( { caps = { calc_server = the_server:svr() },
            log = { "server", "blue" } },
         "rom/server" );

ld:start( { caps = { calc_server = the_server },
            log = { "client", "green" } },
          "rom/client" );
```

`log = { <tag>, <color> }` configures this application's log output. All log messages will be sent to the serial line by default. Every line for the application will start with `<tag> | " and will be colored in the respective VT100 terminal color.
All objects that are placed in a program's capability space can at runtime be requested using

```cpp
L4::Cap<Foo> cap = L4Re::Env::env()->get_cap<Foo>("name");

if (cap.is_valid())
    /* use cap */
```
That's a lot of information!

- There are examples to try out in l4/pkg/examples!
  - Might not be checked out yet, so go to l4/pkg and do
    
    ```bash
    $> svn up examples
    ```
  
  - Example code is rather small and demonstrates only one feature at a time.
Booting Fiasco.OC

- **BIOS** initializes platform and then executes boot sector.
- **Boot loader** (e.g., GRUB) loads kernel and multiboot modules.
  - We skip this step in QEMU.
- **Bootstrap**
  - interprets multiboot binaries
  - sets up the kernel info page
  - Launches Fiasco.OC kernel
- **Fiasco.OC** initializes and then starts
  - **Sigma0** (by definition, L4's basic resource manager)
  - **Moe** (the root task)
Moe starts a new task

- Thread needs code / data / stack to execute.
- Loader retrieves code and data from the program binary.
- Stack is allocated at runtime through the application's memory allocator.

TU Dresden, 2012-07-20
Fiasco.OC: Fundamental Mechanisms
Moe starts a new task

- ELF loader determines binary entry point → EIP
- ELF loader sets up an initial stack (containing argv & env) → ESP
- Loader creates task with initial thread set to EIP / ESP
Initial thread raises a page fault

- **Initial thread starts**
  - accesses first instruction
  - HW finds no valid page table entry
  - page fault!

- **Page faults are a CPU exception (#14)**
  - delivered in kernel mode
  - kernel needs to have set up an IDT entry
  - IDT entry points to a handler function (page fault handler)

- **Page fault handler needs to make a decision on how the HW page table needs to be modified**
  - decision is a policy
    - should be kept out of the kernel!
  - PT modifications are privileged operations
    - needs to be handled in kernel!
Every thread has a unique user-level page fault handler assigned: the pager.
Fiasco.OC: Pager

Application’s address space \rightarrow\text{Fiasco.OC Microkernel} \rightarrow Page Fault Handler \rightarrow \text{IPC call} \rightarrow \text{Pager Code} \rightarrow Pager’s address space

- Page Fault Message: - faulting address - faulting EIP - read/write access

Pager Memory
Fiasco.OC: Memory mapping

- Application’s address space
- Pager’s address space
  - Pager Memory
  - Pager Code

- Fiasco.OC Microkernel
  - Page Fault Handler

Processes:
- resume execution
- map(pages)
- adapt page tables
Page fault reflection and mapping are implemented using IPC.

- Pager blocking waits for page fault messages.
- Range to map from pager to client is described in special data structure: flexpage.

- The thread's UTCB contains an area that is intended to hold flexpages → buffer registers.
• Chicken vs. egg: who resolves page faults for sigma0, the initial pager?
  – Fiasco.OC initially maps all memory to sigma0, so that it never raises page faults.

• User-level pagers allow implementing complex management policies
  – But basic services might only want simple management.

• Solution: Pagers can be stacked into pager hierarchies.
Pager Hierarchies

- Application
- Pager 1
- Pager 2
- Pager 3
- Phys. Memory 1-to-1 mapped
- Initial Address Space
- Disk Driver
- Fiasco.OC
- Microkernel

Fiasco.OC: Fundamental Mechanisms
Pager Hierarchies: A Problem

Initial Address Space

Phys. Memory
1-to-1 mapped

Initial Address Space

Pager 1

Pager 2

Pager 3

Disk Driver

Application

Application

Fiasco.OC
Microkernel
Pager Hierarchies: A Problem

How does the kernel know which pager is responsible for handling a fault?
Pager Hierarchies: A Problem

How does the kernel know which pager is responsible for handling a fault?

Every problem in CS can be solved by another layer of indirection!
How does the kernel know which pager is responsible for handling a fault?

Every problem in CS can be solved by another layer of indirection!
• Every task has a single pager thread, the **region mapper (RM)**.
  – RM is the first thread within an L4Re task.
  – RM maintains a region map:

```
<table>
<thead>
<tr>
<th>Application</th>
<th>VM Region</th>
<th>Pager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region Mapper</td>
<td>&lt;start, end&gt;</td>
<td>Pager 1</td>
</tr>
<tr>
<td>Pager 1</td>
<td>&lt;start, end&gt;</td>
<td>Pager 1</td>
</tr>
<tr>
<td>Pager 2</td>
<td>&lt;start, end&gt;</td>
<td>Pager 2</td>
</tr>
<tr>
<td>Pager 2</td>
<td></td>
<td>Pager 2</td>
</tr>
</tbody>
</table>
```

**Fiasco.OC Microkernel**

**Page-Fault Handler**

**Pager 1**

**Pager Memory**

**Pager Code**
• All threads of a task have the RM assigned as their pager.
• Fiasco.OC redirects all page faults to the RM.
• RM then uses synchronous IPC calls to obtain “real” memory mappings from the external pager responsible for a region.
• This requires a last building block: a generic abstraction for mapping memory.

→ Dataspaces
Dataspaces

- Address spaces may consist of memory from different sources
  - Binary: code/data sections
  - File system: memory-mapped files
  - Anonymous memory (stacks)
  - Network packets
  - Device MMIO regions

- The RM does not care about the details. It only manages regions consisting of pages.
  - generic memory object: dataspace
  - detailed implementation in dedicated memory manager
• Application obtains a **capability** to a **dataspace**

• Application asks its RM to **attach** this dataspace to virtual memory.
  - RM finds free block in virt. AS that fits the DS.
  - RM returns the address of this block.

• The application now uses the dataspace by simply accessing memory.

• The RM makes sure that all page faults are handled by obtaining mappings from the **dataspace manager**.
Putting things together

<table>
<thead>
<tr>
<th>VM Region</th>
<th>Dataspaces</th>
</tr>
</thead>
</table>
| <start, end> | \textit{Dataspace Manager 1, 1}  
| <start, end> | Dataspase Manager 2, 1  
| <start, end> | Dataspase Manager 2, 2  

Application's Address Space

Dataspase Manager 1
Dataspase 1

Dataspase Manager 2
Dataspase 1

Dataspase Manager 2
Dataspase 2

Dataspase Manager's Address Space

Dataspase Manager's Memory

Region Map

Dataspaces

DATASPACE MANAGER 1
DATASPACE 1

DATASPACE MANAGER 2
DATASPACE 1

DATASPACE MANAGER 2
DATASPACE 2
Assignment

• Implement an encryption server based on L4Re
  – For an example, see l4/pkg/examples/clintsrv

  – 2 functions: encrypt and decrypt
    • Encryption: client sends plain text and receives encrypted version.
    • Decryption: client sends encrypted text and receives plain text.

  – If you fancy learning about dataspaces:
    • initialize session by letting the client pass a dataspace to the server
    • The dataspace can then be used to exchange plain and encrypted texts
Further Reading

- N. Feske: “A case study on the cost and benefit of dynamic RPC marshalling for low-level system components”
  http://os.inf.tu-dresden.de/papers_ps/feske07-dynrpc.pdf
  Using C++ streams for IPC marshalling

  http://cgi.cse.unsw.edu.au/~kevine/pubs/acsac01.pdf
  Generic memory management using dataspaces

- G. Hunt: “Singularity – Rethinking the Software Stack”
  http://research.microsoft.com/apps/pubs/?id=69431
  Some very cool ideas on using safe languages for implementing operating systems.