M3 – Microkernel for Minimalist Manycores

Nils Asmussen

MKC, 06/19/2014
Outline

1. Introduction
2. Spatial Isolation
3. Abstractions
4. Summary
Motivation

- We are moving towards accelerator architectures
- But how to really integrate heterogeneous cores and provide isolation?
- Energy consumption becomes more and more important
- Experiments show, that caches and MMUs consume lots of energy
- We try to address these issues by supporting heterogeneous cores, provide isolation, but be less energy demanding
Vision

Global Memory

Core 0: App1
- PE
- Local Mem
- DTU

Core 1: µ-kernel
- PE
- Local Mem
- DTU

Core 2: App2
- PE
- Local Mem
- DTU

Core 3: NET
- PE
- Local Mem
- DTU

Core 4
- FPGA
- Local Mem
- DTU

Core 5
- FPGA
- Local Mem
- DTU

Core 6
- h264
- Local Mem
- DTU

Core 7: FS
- PE
- Local Mem
- DTU

Network

Storage
Outline

1. Introduction
2. Spatial Isolation
3. Abstractions
4. Summary
Isolation Principle

Global Memory

- app
- μk
- app
- app
- app
- app
Data Transfer between Cores
Memory Access

Core A
- PE
- DTU
- MEM

Core B
- PE
- MEM
- DTU

Global MEM
- data

Channel
- base
- total
- ro
donly
Outline

1 Introduction
2 Spatial Isolation
3 Abstractions
4 Summary
Capabilities

- The $\mu$-kernel manages the capabilities of all tasks
- Like in L4, capabilities are protected by the kernel, selectors are managed by the applications
- Normal IPC can't be used because the kernel is not involved
- Thus, a special protocol is used to exchange capabilities
Sessions

Introduction

Spatial Isolation

Abstractions

Summary

sessions

createsrv(0,"fs")

createsrv(0,"keyb")

sysc gate
gate cap
service cap
session cap

App

Kernel

FS

Keyb

App

Kernel

DTU

fs

keyb

app

kernel
Spatial Isolation

Abstractions

Summary

Sessions

createsrv(0,"fs")

createsrv(0,"keyb")

App

Kernel

DTU

FS

Keyb

App
gate cap

service cap

session cap
Introduction

Spatial Isolation

Abstractions

Summary

Sessions

```
createsess(0,"keyb")
createsess(1,"fs")
```
Sessions

FS

Keyb

App

Kernel

createsess(0,"keyb")

createsess(1,"fs")

DTU

sysc gate
gate cap
service cap
session cap

App
Keyb
FS
Sessions

- FS
- Keyb
- App
- Kernel
- DTU

creategate(2,1,lbl)
Sessions

- FS
- Keyb
- App
- Kernel
- DTU

creategate(2,1,lbl)

- sysc gate
- gate cap
- service cap
- session cap
Sessions

FS

Keyb

App

Kernel

DTU

delegate(0,2)
debute(0,2)

response: 1
Sessions

- FS
- Keyb
- App
- DTU

delegate(0,2)
delegate()
Sessions

Introduction
Spatial Isolation
Abstractions
Summary

- FS
  - delegate(0,2)
  - response: 1
- Keyb
  - delegate()
- App
  - delegate(0,2)
- Kernel
  - DTU
  - FS
  - Keyb
  - App
Sessions

FS

Keyb

App

Kernel

DTU

obtain(1,3)

response: 1

gate cap

service cap

session cap

sysc gate
Sessions

![Diagram showing the Sessions structure with interactions between FS, Keyb, App, and DTU]

**Observe:**
- FS interacts with Keyb via `obtain()`.
- App obtains a session with `obtain(1,3)`.
- DTU is connected to the other components.

Key components and interactions:
- `obtain()` function is used for acquiring resources.
- Sessions are established through specific interactions between the application and the kernel.

**Summary:**
- Spatial isolation and abstractions are crucial in managing these interactions.

**Diagram Elements:**
- FS (File System)
- Keyb (Keyboard)
- App
- DTU (Device Tree Unit)
- Gate cap
- Service cap
- Session cap
- Sysc gate

**Note:** The diagram illustrates how sessions are managed and isolated within the system architecture.
Sessions

- FS
- Keyb
- App
- DTU

Flow:
- App → Kernel: obtain(1,3)
- FS → Kernel: obtain()

Abstractions:
- sysc gate
- gate cap
- service cap
- session cap
Sessions

- FS
- Keyb
- App
- DTU

**Obtain()**

- response: 1
- obtain(1,3)
Sessions

- FS
- Keyb
- App
- DTU

Obtain(1,3)

Obtain()

Response: 1

Obtain(1,3)
Sessions

1. FS
2. Keyb
3. App
4. Kernel

Symbols:
- FS: File System
- Keyb: Keyboard
- App: Application
- Kernel: Operating System Kernel
-SySC Gate
-Gate Cap
-Service Cap
-Session Cap

Example:
- Key pressed
- Open(/foo)
- Response
Sessions

- FS
- Keyb
- App
- Kernel

open("/foo")

key pressed

sysc gate

gate cap

service cap

session cap
Sessions

- **FS**
- **Keyb**
- **App**
- **Kernel**
- **DTU**

**Open Command:**
```
open("/foo")
```

**Response:**
```
sysc gate
```

**Gate Cap:**
```
gate cap
```

**Service Cap:**
```
service cap
```

**Session Cap:**
```
session cap
```
The syscall `reqmem` can be used to request global memory.

Global memory is managed by the kernel.

This memory capability can be put on a channel to access it.

For that reason, memory capabilities are the same as gate capabilities.

I.e. unlike L4, there are no different kinds of capabilities.

Syscall `derivemem` to create a memory capability for a subset of a given one.

Syscall `createmem` to create a memory capability for the own local memory.
There is a filesystem service running on some core
The FS uses global memory as the inode and buffer cache
App goes to FS in order to know where file fragments are in global memory
FS hands out memory capabilities to the application
App reads/writes directly from/to global memory afterwards
Filesstem – API

- Like in L4Re, the mount-tree is application local
- The File abstraction is the low level abstraction to read/write files
- The BufferedFile abstraction sits on top and provides a buffer in between
Like in L4Re, the mount-tree is application local

The File abstraction is the low level abstraction to read/write files

The BufferedFile abstraction sits on top and provides a buffer in between

Example

```cpp
VFS::mount("/", "m3fs");

BufferedFile file;
VFS::open("/myfile", VFS::R, file);
while(file.read(buf, 512) > 0) {
    // ...
}
```
Tasks

- Syscall `createtask` to create a new task
- The kernel assigns the task to a core, if there is a free one
- The application gets the task capability and a memory capability for the whole local memory of that core
- Thus, it has complete control over that core
- Syscall `start` to start the task
- It can exchange capabilities with the task before starting it
- Syscall `wait` to wait until a task is finished (called `exit`)
Tasks – Examples

Executing ELF-Binaries

```cpp
Task task("test");
char *args[] = {"/bin/hello", "foo", "bar");
task.exec(3, args);
```
Tasks – Examples

Executing ELF-Binaries

```cpp
Task task("test");
char *args[] = {"/bin/hello", "foo", "bar"};
task.exec(3, args);
```

Asynchronous Lambdas

```cpp
Task task("test");
int a = 10;
task.delegate(CapRngDesc(1, 2));
task.run([a](){
    Serial::get() << "a=" << a << "\n";
});
```
M3 is a different kind of microkernel-based system

- The $\mu$-kernel manages remote resources instead of local resources
- The $\mu$-kernel and applications run alone on the cores
- Applications are linked against a library to get abstractions
- Capabilities are used to manage permissions in the system
- Similar operations with caps, but still different