

# MALDOS:

## a Moderately Abstracted Layer for Developing Operating Systems

Mattia Maldini

Relatore: Renzo Davoli

Università di Bologna

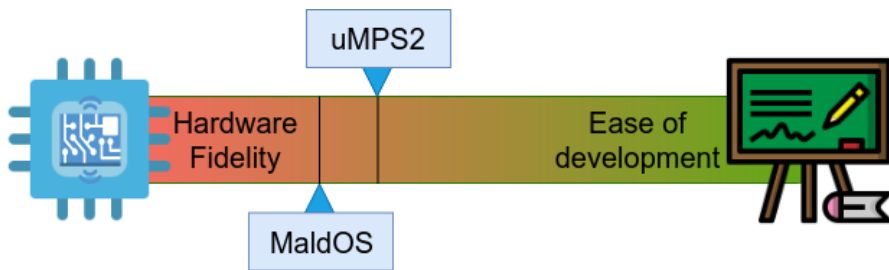
Sessione di laurea 14 marzo 2019

Structuring an Operating Systems course can be a thorny task.

Topics like scheduling, parallel programming, virtual memory and context switching are hard to grasp via a purely abstract approach, and developing a concrete example is inherently non trivial.

Examples might be implemented as user space programs, but they would lose some details that deeply characterize kernel development. On the other end one could work, bare metal, on a real processor, but this entails many nuances unnecessary for students.

A previous solution to this issue was the  $\mu$ MPS family of emulators; this work proposes a slight shift in the form of an abstraction layer allowing for easier development on real hardware.



$\mu$ MPS2 allows students to work in a simplified environment while still being faithful to the real MIPS architecture, providing an interesting experience. However

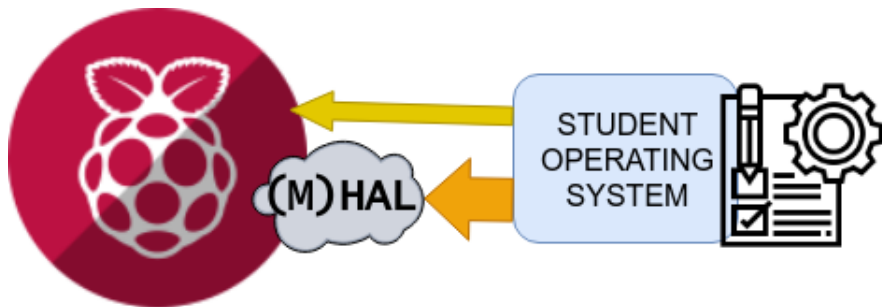
- it is still abstract work, as the resulting OS will never run on a real device
- $\mu$ MPS is an ad-hoc emulator
- Its debugging facilities lack step-by-step execution of C code
- MIPS is a dated architecture

**MaldOS** is an abstraction layer devised to reproduce an  $\mu$ MPS-like simplified environment on a real device and architecture, the Raspberry Pi 3 and ARMv8.

- it is an extremely practical example, and the result can be run on a real machine
- it relies on general purpose software (namely, Qemu and GDB)
- its debugging facilities are less specific but more powerful (full GDB support)
- ARM is a more modern and widespread environment

# Bare Metal

The problem with a real architecture is the intrinsic complexity of hardware management. The abstraction layer softens the approach by partially taking care of initialization and configuration procedures.



- A proper linker script is setup beforehand
- Move to the correct level of execution
- Prepare memory structure (stack pointers and such)
- Boot all four kernels and park three
- Prepare the C environment (zero bss section)
- Configure and initialize all real devices (UART, screen, SD)

# Interrupt Management

EL3

Security-dedicated level, uninteresting for the purpose of this work

EL2

## MaldOS handler

EL1

## MaldOS handler

Student OS

ELO

User  
process



**MaldOS** creates  $\mu$ MPS-style emulated devices with an ideal interface. Thanks to the mailbox interface the integration is seamless.

| Virtual Disk device registers |
|-------------------------------|
| status register               |
| command register              |
| data 0 register               |
| data 1 register               |
| mailbox register              |

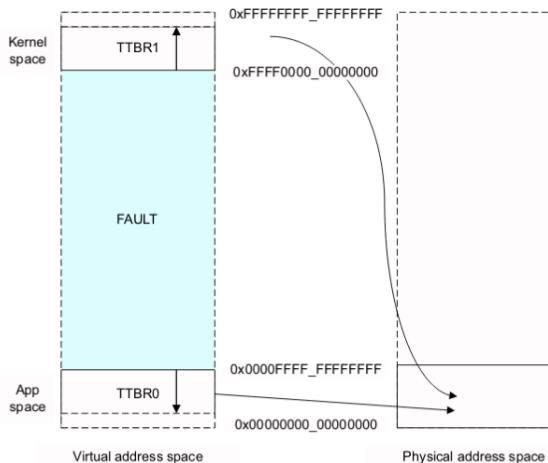
| EMMC device registers     |
|---------------------------|
| second argument register  |
| block size count register |
| first argument register   |
| command register          |
| response registers (4)    |
| status register           |
| control register (2 of 3) |
| interrupt register        |
| interrupt mask register   |

... 11 more

# Virtual Memory

The ARMv8

MMU is remarkably flexible, so the virtual memory configuration is left mostly untouched. Students are assisted by changing page tables every time the kernel context switched in.



# Qemu and Debugging

Load Binary

/path/to/target/executable -and -daps

hide filesystem

fetch disassembly

reload file

jump to line

/home/makus/Projects/test/uARMv2/example/source/app\_main.c:133 (238 lines total)

Fetch source files

Enter file path to view, press enter

Expand all Collapse all

Reveal current file

23 known files used to compile the inferior program

root

home

makus

Projects

test

uARMv2

example

source

app\_main.c

include

source

usr

lib

124

}

125

}

126

}

127

void test1() {

128

int numeri[100];

129

unsigned char buffer[5000];

130

tapereg\_t tape\_reg = {0, READ\_REGISTERS, 0, 0, 0};

131

}

132

int contatore = 0;

133

print("partenza 1\n");

134

}

135

CORE0\_MAILBOX0 = (((uint32\_t)(uint64\_t)tape\_reg) & ~0xF) | 0x4;

136

while (tape\_reg.mailbox == 0)

137

nop();

138

}

139

tape\_reg.data0 = (unsigned int)(unsigned long)buffer;

140

tape\_reg.command = READBLK;

141

CORE0\_MAILBOX0 = (((uint32\_t)(uint64\_t)tape\_reg) & ~0xF) | 0x4;

142

while (semaphore == 0)

143

;

144

}

145

print("letto nastro: ");

146

print((char \*)buffer);

147

print("\n");

148

}

149

buffer[0] = buffer[0] == 'M' ? 'P' : 'M';

150

}

151

semaphore = 0;

152

tape\_reg.command = WRITEBLK;

153

CORE0\_MAILBOX0 = (((uint32\_t)(uint64\_t)tape\_reg) & ~0xF) | 0x4;

154

while (semaphore == 0)

155

;

156

}

157

print("scritto nastro\n");

158

}

selected Thread 2, id 2, stopped

| func           | file             | addr    | args |
|----------------|------------------|---------|------|
| idle_core_spin | source/int.S:368 | 0x81150 |      |

selected Thread 3, id 3, stopped

| func           | file             | addr    | args |
|----------------|------------------|---------|------|
| idle_core_spin | source/int.S:368 | 0x81150 |      |

selected Thread 4, id 4, stopped

| func           | file             | addr    | args |
|----------------|------------------|---------|------|
| idle_core_spin | source/int.S:368 | 0x81150 |      |

local variables

+ buffer unsigned char [5000]

contatore 0 int

+ numeri [100] int [100]

+ tape\_reg [...] tapereg\_t

status 0 unsigned int

command 6 unsigned int

data0 0 unsigned int

data1 0 unsigned int

mailbox 0 unsigned int

expressions

expression or variable

no expressions in this context

Tree

width (px)

height (px)

create an Expression, then click when viewing a variable with children to

Type "apropos word" to search for commands related to "word".

0x0000000000000000 in ?? ()

set breakpoint pending on

Thread 1 hit Breakpoint 1, main () at source/app\_main.c:221

221 print("sono l'applicazione")

");

Thread 1 hit Breakpoint 2, test1 () at source/app\_main.c:133

133 print("partenza 1")

");

(gdb) enter gdb command. To interrupt inferior, send SIGINT.