RTOS, RTAI
μ-kernel and Fiasco

Jonas Flodin
Today's lecture

- RTOS
- RTAI
- μ-kernels
- L4/Fiasco
- Example
- Memory management exercise
Real-time Systems

- Processing must meet hard timing constraints
- Examples
  - Sensor writes to finite buffer
  - Drive/Fly-by-wire
- Timeliness is a criteria for correctness
Real-time Tasks

- Abstraction of a piece of SW
- Scheduling unit
- A collection of timing information
  - Inter release time
  - Relative deadline
  - Worst Case Execution Time (WCET)
Response Time

- Delay between release and completion of a task
- If response time is smaller than deadline for all tasks, system is **schedulable**
- Worst case response time calculation is a sum of all possible delays
Response Time

\[ \tau_1 \quad \tau_2 \]

Blocked

Response Time

Finish
What affects response time?

- **WCET**
  - Program code
  - Architecture
  - System calls
  - Hardware state

- **WCET of higher priority tasks** (fixed priority scheduling)

- Waiting for critical sections/semaphores

- Interrupt latency
Real-time Operating System (RTOS)

- Minimize worst case delays
  - Bounded delays
- Deterministic behaviour
  - Strict scheduling
  - Static memory
- Limited functionality
- Preemptive kernel
Some RTOS issues

- **Hardware**
  - DMA devices
  - Caches
  - Interrupts

- **Software**
  - System calls
  - Semaphores
  - Memory management
  - Language
Hardware

- DMA devices
  - Might block execution indefinitely
  - TDMA is a costly solution

- Caches
  - Speeds up memory fetches
  - Hard to predict
  - Some systems disable caches
Interrupts

- Interrupt latency
  - Longest non-preemptive section
  - Adds to response times

- Excessive amounts of interrupts are bad
  - Interrupt handlers are highest priority
  - Unacceptable delay for some control applications
  - Solutions are:
    - Deferred interrupts
    - Periodic interrupt polling
System calls

- Preemptive system calls are necessary
- Upper bound on execution time
- Ex: inter process communication
  - Implemented with message queues
  - Dispatched in priority order
Memory and Semaphores

- Static memory management
- Segmented memory
- Dynamic memory management is possible, but not widely used

- Use of semaphores needs special attention
  - Priority inversion must be handled
  - As small as possible sections
Languages

- Most common is C/assembly
- Some industries favour ADA
- High level languages unsuitable
  - Hard to predict performance
  - Ex: Java
Standards

- There are many standards for real-time systems
  - RT-POSIX – Real-time extension to POSIX
  - OSEK/VDX – Automotive
  - ARINC 653 – Avionics
  - Micro-ITRON – Consumer electronics
- Defines APIs and requirements
Is Linux real-time?
Is Linux real-time?

- No!
  - System calls (fork, malloc)
  - Paging
  - Fairness
  - Reordering
  - Batching
RTAI

- Real-time Linux
- Kernel executed as a low priority task
RTAI

- RT-interrupts are serviced immediately
- Non-RT-interrupts are handled by Linux kernel
- Fully preemptible kernel
- Determinism at the cost of additional delay when intercepting interrupts
μ-kernel

L4/Fiasco
μ-kernel

- Very much like the name suggests

**Advantages**
- Small kernel (LoC and binary size)
- Less room for errors
- Smaller attack surface
- Isolation

**Drawbacks**
- Few features
- Performance
Monolithic VS. μ-kernel

- Monolithic: 2 kernel entries/exits
- μ-kernel: 4 kernel entries/exits + 2 context switches
L4

- Originally written by Jochen Liedtke
  - Purely optimized x86 assembly
  - Focused on performance
  - Only 12KB

- Adapted over time
  - Maintainability
  - Portability
  - Higher level languages
L4

All flavours of L4 have one common spirit:

*Only put the absolutely necessary functionality in the kernel, all other functionality shall be implemented on top of the kernel in user applications.*

*No policy in the kernel, the kernel shall only implement mechanisms.*
Fiasco

- One flavour of L4
- Developed at TU Dresden
- Written in C++ (object oriented)
- Runs on x86 and ARM
- SMP support
- Virtualization support
Capabilities

- Tokens for access rights
- Managed by kernel
- Used for:
  - Memory
  - IPC
  - Tasks
  - I/O
  - IRQ
  - Factories
L4Re

- L4 Runtime environment
- Rich in features
- Libraries
- Makes life easier for programmer
Software stack

- User applications
  - L4Re
  - μ-kernel

- Hardware
- Fiasco
- Root pager
- Root task
- Libraries
- Task

- non-privileged mode
- privileged mode
System Startup

- BIOS
- Bootloader
- Bootstrapper
- Kernel
- Userlevel tasks
BIOS

- Basic Input/Output System
- Firmware
- Started at power on
- 3 main functions
  - Test hardware
  - Provide hardware abstraction
  - Start second-stage boot loader
- Old, but is being reworked
Second-stage boot loader

- Loads operating system
- Specifies parameters to OS
  - Fail-safe/recovery mode
  - etc.
- User interaction
- GNU GRUB
Example Fiasco Entry

```c
menuentry 'hello world' {
    multiboot (hd0,1)/boot/fiasco/bootstrap -serial -comspeed 115200 -comport 1
    module --modaddr 0x01100000 (hd0,1)/boot/fiasco/fiasco /fiasco -serial_esc -jdb_cmd=JH -wait
    module (hd0,1)/boot/fiasco/sigma0 /sigma0
    module (hd0,1)/boot/fiasco/moe /moe --init=rom/hello
    module (hd0,1)/boot/fiasco/l4re /l4re
    module (hd0,1)/boot/fiasco/hello /hello
}

menuentry 'Graphics' {
    multiboot (hd0,1)/boot/fiasco/bootstrap -serial -comspeed 115200 -comport 1
    module --modaddr 0x01100000 (hd0,1)/boot/fiasco/fiasco /fiasco -serial_esc -wait -jdb_cmd=JH
    module (hd0,1)/boot/fiasco/sigma0 /sigma0
    module (hd0,1)/boot/fiasco/moe /moe rom/graphics.cfg
    module (hd0,1)/boot/fiasco/l4re /l4re
    module (hd0,1)/boot/fiasco/ned /ned
    module (hd0,1)/boot/fiasco/graphics.cfg /graphics.cfg
    module (hd0,1)/boot/fiasco/graphics-server /graphics-server
    module (hd0,1)/boot/fiasco/graphics-client /graphics-client
    module (hd0,1)/boot/fiasco/kbd /kbd
    module (hd0,1)/boot/fiasco/io /io
    module (hd0,1)/boot/fiasco/fb-dr v /fb-dr v
    module (hd0,1)/boot/fiasco/x86-legacy.devs /x86-legacy.devs
    module (hd0,1)/boot/fiasco/x86-fb.io /x86-fb.io
    module (hd0,1)/boot/fiasco/l4lx-x86.io /l4lx-x86.io
}
```
Bootstrap

- Takes over after boot loader
- Sets up memory regions
- Copies modules
- Leaves control to Fiasco kernel
Fiasco (kernel)

- Keeps track of capabilities
- Implements mechanisms
  - Virtual memory
  - Scheduling
  - IPC
  - Factories
- Starts root pager and root task
Sigma0 – root pager

- Initially gets all memory
- FCFS
- Page granularity
- Hierarchical memory
Hierarchical memory

- RAM
- Caps
- Sigma0
- Task A
- Task B
Moe and Ned

- Moe
  - Root task
  - Starts other tasks

- Ned
  - Lua script parser
  - Loads programs
  - Sets up capabilities
Example: Client – Server

- Configuration in Lua script

```lua
-- Local name for Moe
local ld = L4.default_loader;

-- Create a new IPC gate
echo_server = ld:new_channel();

-- Start the server passing the factory channel
ld:start( { caps = {factory = echo_server:svr()},
            log = {"server", "green"},
            "rom/server-binary" } );

-- Start the client
ld:start( { caps = {echo_server = echo_server:create(L4.Proto.Ipc_gate,"")},
            log = {"client", "blue"},
            "rom/client-binary MyString" });
```
C++ code

Server

```cpp
int Echo_server::dispatch(
    l4_umword_t, L4::Ipc::Iostream &ios)
{
    l4_msgtag_t tag;
    char buf[200];
    ios >> tag;
    switch(tag.label()) {
    case L4::Meta::Protocol:
        ...
    case L4::Factory::Protocol:
        ...
    case Protocol::Print_string:
        read_text(ios,buf);
        printf("%s", buf);
        ios << Protocol::OK;
        return L4_EOK;
    }
    return -L4_EBADPROTO;
}

int main(int argc, char ** argv){
    static Echo_server myServer;
    if (!server.registry()->register_obj(
        &myServer,"factory").is_valid())
        return 1; // error
    server.loop(); // Wait for requests
    return 0;
}
```

Client

```cpp
void show(const char * text){
    L4::Cap<void> server = L4Re::Env::env()->
        get_cap<void>("echo_server");
    if (!server.is_valid())
        return;
    L4::Ipc::Iostream s(l4_utcb());
    s << text;
    int r = l4_error(s.call(server.cap(),
        Protocol::Print_string));
    if (r){
        return; // error
    s >> r;
    if(r == Protocol::OK)
        return; // OK!
}

int main(int argc, char ** argv){
    if(argc < 2)
        show("Hello world!");
    else
        show(argv[1]);
    return 0;
}
```
Demo
Memory management assignment

- User level memory management
- `malloc()` and `free()`
How it works

Task A virtual address space

Task A

malloc() | initialize_mem() | Env

allocate() | attach()

RAM
How it works

Task A virtual address space

Task A

malloc() initialize_mem() allocate()

Bookkeeping
Next = NULL
Prev = NULL
Size = DEFAULT
Allocated = FALSE

Unallocated Space

Env

RAM
How it works

Task A virtual address space

Task A

malloc() initialize_mem()

allocate() attach()

init data structure

perform allocation

Env

RAM
Perform allocation

VA

NULL

PA

Next
Prev
Free
Size

NULL
Perform allocation
Perform allocation
Growing

- Eventually we'll run out of space
- Just ask for more memory
Growing

Task A

malloc() -> grow() -> allocate() -> attach() -> update datastructure

perform allocation

Env

PA
Deallocating

VA

free()

free()

unused

used
Questions?