L4 Development using Iguana

Ben Leslie

Ben.Leslie@nicta.com.au

Embedded, Real-Time and Operating Systems Program
National ICT Australia
August 2004

OVERVIEW

Last time we covered the basics of getting a simple L4 application running:

- ➜ Tools required
- ➜ Configuring and building a kernel
- ➜ Creating a boot image using dite
- ➜ Loading and kernel initialisation
- ➜ Basic use of the L4 kernel debugger
- ➜ Interaction between sigma0 and root task during start up

This time: Building and programming in Iguana

- ➜ SCons build tool and the Iguana build system
- ➜ Boot image generation and Wombat booting
- ➜ Writing device drivers

IGUANA PROJECT

Source code layout:

- libs – library implementation
  - Includes driver, l4, l4e, drv_sa1100_uart
  - Generally have a `src/` and `include/` directory
- apps – application, service implementation
  - Generally have a `src/`
- l4linux – Wombat source code
- pistachio – L4 kernel code
- tools – Script used during build
- build – Output of build

BUILD PROCESS

Iguana uses the SCons build tool.[a]

> *SCons is an Open Source software construction tool – that is, a next-generation build tool.*

SCons is written in Python and the build scripts are actual python files – allows arbitrary scripting.

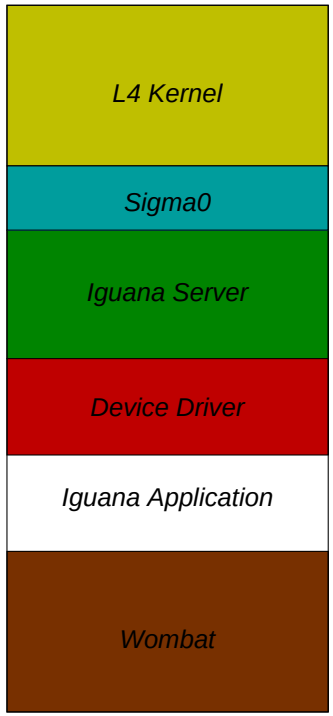
SCons comes with rules for building standard libraries and applications.

The Iguana build system provides rules for building L4 based projects.

[a] dite also uses this

BUILD SYSTEM CONCEPTS

The aim of the build system is to produce a bootimage, which will be constructed of a set of applications.



Bootimage

Environments are used to group together applications. Environments share:

- ➜ Libraries
- ➜ Compiler and compiler flags

Libraries are specified as being part of a particular environment. Libraries have options that can be specified per environment. E.g: C library.

Project is specified in one top-level file (`SConf`).

Component build information is specified in the component directory. E.g.: `apps/iguana/SConstruct` or `libs/c/SConstruct`.

Pistachio and Linux both have existing build systems that we have not tried to replicate. Rather we use SCons to call out to the existing build systems.

Build system walk-thru:

```
$ scons
```

After some time, assuming no errors, this should produce a bootable image: `build/bootimg.dite`

IGUANA AND WOMBAT BOOTING

Last time we saw the boot sequence up to the root-server.

This time we look at Iguana booting up to Wombat startup.

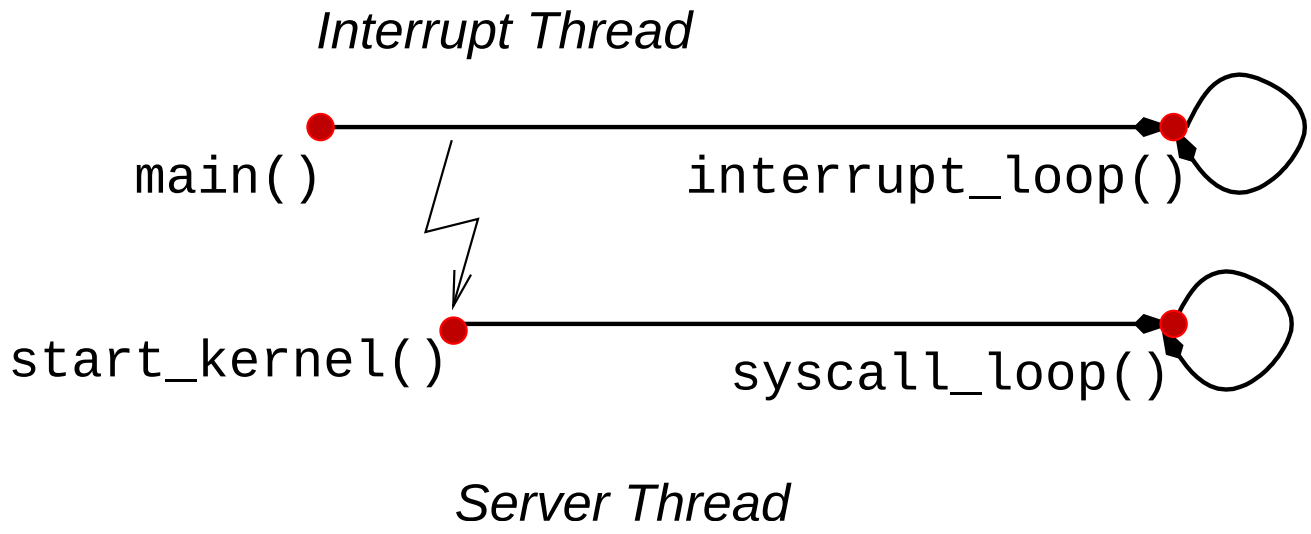
1. Iguana startup – `apps/iguana/src/main.c:main(void)`
   ① `setup_vm()` – Initialise page tables, frame allocator.
   ② `kmalloc_init()` – Initialise kernel memory allocator.
   ③ `objalloc_init()` – Initialise the memory section allocator.
   ④ `objtable_init()` – Initialise the memory section table.
   ⑤ `pd_init()` – Initialise the protection domains structures.
   ⑥ `populate_init_objects()` – Add initial objects (page table and bootimage).
   ⑦ `utcb_init()` – Initialise an area of the SAS to store UTCBs.
   ⑧ `thread_init()` – Initialise thread allocator.
   ⑨ `start_init()` – Find and start the init task
   ⑩ `iguana_server()` – Go into server loop servicing page faults and requests.

2. Iguana init – Equivalent of `init` in Linux. It is **not** another server!
   ① Start serial driver.
   ② Initialise serial device.
   ③ Start Wombat server.
   ④ Set serial stream $\Longrightarrow$ wombat server.

This startup is obviously not ideal since it is hard coded. Currently looking at a suitable scripting language so that init parses a configuration script. Possibilities are:

- ➜ lua
- ➜ python
- ➜ home grown "conf".

3. Wombat startup – `l4linux/arch/l4/kernel/main.c:main()`
   ① Spawn lower priority *server thread*.
   ② Continue on as the *interrupt thread*.
   ③ *Server thread* starts Linux's arch independent `start_kernel()`.



WRITING DEVICE DRIVERS

Attempt to reduce burden of device driver writing by restricting the *device driver* to just handling hardware, and leaving policies such as queuing to generic libraries.

SA1100 uart driver will be presented as a simple example.

DDDSL

Device Driver Domain Specific Language.

Specify a device's properties — in particular, register layout — in a high level language.

UART example:

```
UTCR3 32 @ 0xc:
 rxe <0>        # Receive enable
 txe <1>        # Transmitter enable
 brk <2>        # Break
 rie <3>        # Recevie FIFO interrupt enable
 tie <4>        # Transmit FIFO interrupt enable
 lbm <5>        # Loopback mode
```

This information is stored in `name.reg` files and compiled by the build system into `name.reg.h` and `name_types.reg.h`

The `name.reg.h` file provides a set of inline function for accessing the device registers.

THE UART DRIVER

The UART provides a stream interface and as such implements the character device operations:

```
static struct character_ops ops = {
        /* Driver ops */
        { setup,
          enable,
          cleanup,
          interrupt },
        /* Character ops */
        write,
        read
};
```

This sets up a simple indirection table. All a driver is required to do is implement each of the above functions.

IS THIS RELEVENT FOR AMSS DRIVERS

- ➜ Not sure of interfaces required..
- ➜ Reuse in different environments not required..

So lets look at the low-level access

IGUANA DEVICE ACCESS

Access device registers:

```
memory = memsection_create(0x1000, (uintptr_t*)((unsigned long)&space));
hardware_back_memsection(memory, 0x80050000, 1);
```

This may be done in the driver itself, or externally and pass the pointer in.

Interrupts:

```
hardware_register_interrupt(L4_Myself(), 17);
```

Again, maybe done external to the driver.

Support for DMA:

Iguana provides `pin_range` method to support direct memory access.

Returns a scatter-gather list of physical pages.

