Embedded, Real-Time and Operating Systems Program
http://nicta.com.au/ertos.html

Iguana User Manual

Gernot Heiser
gernot@nicta.com.au

DRAFT Document Revision: 1.62 Date: 2005/04/04 01:01:32

Abstract

This document describes the Iguana embedded operating system. It introduces the basic concepts of Iguana and describes the API in an abstract, OO-style notation, as well as providing specific language bindings and example code.

Contents

Chapter 1

Introduction

Iguana is designed as a base for the provision of operating system (OS) services on top of the L4 microkernel [**?**], specifically the Version 4 API [**?**] as implemented by L4Ka::Pistachio [**?**].

Furthermore, Iguana is designed for use in embedded systems. The implications of this are:

- Iguana complements, rather than hides, the underlying L4 API. It provides services virtually every OS environment requires, such as memory and protection management, and a device driver framework;
- the memory and cache footprints of Iguana are kept small;
- low-overhead sharing of data is supported;
- Iguana attempts to provide the best possible performance on typical embedded processors. In particular, it supports the separation of protection and translation that is a feature of some embedded processors, such as ARM cores, by encouraging a non-overlapping address-space layout.

The objectives of low-overhead sharing and non-overlapping address-space layout are supported by allowing separate processes to securely share a single address space. This follows the idea of a *single-address-space operating system* (SASOS), such as Angel [**?**], Opal [**?**] or Mungi [**?**].

Unlike Opal or Mungi, Iguana does not assume that all data in the system, including persistent data (stored on disk or FLASH) resides at an immutable address in the single address space. Similar to Nemesis [**?**], Iguana uses the single address space only for data that presently resides in primary memory. Therefore, there is no guarantee that the address of data will not change once it leaves primary memory. This is essential to supporting 32-bit architectures.

Another difference to single-address-space operating systems is that Iguana does not force the single-address-space view onto applications. Applications have a choice of creating new address spaces as in other systems, but can also create processes that share the creator's address space, although firewalled into a separate *protection domain*. This allows the application designer to trade off performance (on some architectures) and simplicity of sharing against the amount of available address space (on 32-bit architectures) and the ease of porting legacy code [**?**].

Iguana's programming model borrows heavily from Mungi [**?**], but makes a number of simplifications with respect to Mungi. The similarity goes in fact beyond API issues: Iguana and Mungi share a significant amount of code. However, while Mungi is designed for 64-bit processors, Iguana supports 32-bit as well as 64-bit hardware. The main hardware requirement for running Iguana is a *memory-management unit* (MMU), or at least a *memory-protection unit* (MPU). In the following we will refer to an MPU when only the protection aspect is relevant, and to an MMU when support for independent virtual-address mappings are required.

The purpose of this document is to explain the use of Iguana for building L4-based embedded systems. It assumes familiarity with L4 and its concepts; a good source of such information is the L4 User Manual [**?**].

Chapter 2 provides an overview of the concepts and basic mechanisms provided by Iguana.

Chapter 3 gives a short description of the Iguana API. There is also discussion on the client server model used by Iguana, as well as some explanation of the way capabilities are used in the system.

Chapter 4 describes all aspects of Iguana's protection system. It also provides some examples of how Iguana can use the capability system to produce certain protection schemes.

Chapter 5 details resource management of the system. It describes the model used and how each of the resources are charged according to this model.

Chapter 6 describes the services provided by Iguana.

Chapter 7 outlines the process the Iguana system goes through during system start up. It also gives details on the role of σ_0 in start up and how to build a boot image.

Appendix A ***will describe after appendix A is complete***

Appendix B details the applications required to build the Iguana system. Also gives instruction on where to obtain the Iguana sources as well as a set of build instructions.

Appendix C shows a commented example of how to use the Iguana system to create something in the Iguana system which is equivalent to creating a process in a Unix based system.

Appendix D lists all the current implementation restrictions that are imposed by the current implementation of Iguana, L4 or some other system constraint.

Appendix E gives a listing of the C bindings used in Iguana Library API.

Appendix F ***will describe after appendix F is complete***

Appendix G prints out Iguana's main IDL file.

Chapter 2

Overview of Iguana Concepts

Iguana introduces the following concepts and abstractions: *protection domain* (PD), *thread*, *session*, *memory section*, *capability*, and *external address space* (EAS). Their meaning is as follows.

Threads are the unit of execution/scheduling. Iguana threads are L4 threads and are manipulated directly with L4 primitives (e.g. *ExchangeRegisters()*) as well as primitives defined by Iguana.

Memory sections are the unit of virtual memory allocation and protection. A memory section is a contiguous range of virtual pages. A memory section can only be de-allocated as a whole, and is homogeneous with respect to protection: a thread having certain (*RWX*) access rights to a particular location in a memory section has exactly the same access rights to any other location in the same memory section. Access to memory sections may be shared between threads in separate PDs. Explicit L4 mapping IPC is not permitted between Iguana protection domains.

Protection domains provide memory protection between threads executing different programs. A PD roughly corresponds to the concept of a *task* or *process* in other systems. However, unlike processes in systems like Linux, PDs all share the same virtual address space (the *Iguana address space* (IAS)). A PD contains zero, one or more threads.

Threads in the same PD have full access to each other's memory, while threads in different PDs are protected from each other, and can access each other's memory only if permitted by the protection system (i.e., if they hold the right capabilities).

Threads inside a PD can create other PDs or EASs (if they hold appropriate capabilities, discussed below).

A PD is created by calling a creation method on an existing PD. This previously existing PD is the new PD's *owner*. The owner of an object is the entity that is charged for the resources used by the object (see Chapter 5).

Communication between PDs is accomplished using *sessions*, described below.

Capabilities are security tokens that define access rights to objects (memory section, thread, PD, EAS). In order to invoke a method on an object, the invoker must hold an appropriate capability. Capabilities are stored in a user-managed system data structure where the system locates them as needed. Hence capability presentation is *implicit* — methods do not have capability arguments. As a consequence, most applications never need to deal with capabilities explicitly.

Certain operations are not controlled by capabilities, but are allowed at any time. These include some L4 primitives on certain objects: local thread manipulation (*ExchangeRegisters()*), IPC to local threads (within the same PD or EAS) and IPC between a thread and its owner.

Sessions are established to allow communication between two protection domains. Sessions are destroyed either explicitly, or whenever either of the protection domains is destroyed. When a session is destroyed, both participants (or which ever one still exists) are notified by Iguana invoking a notification method, allowing the PD to clean up any resources allocated to a session. Currently the only capability required to create a session between two protection domains is the master capability, i.e., there is no restrictions on who you can create a session with at present.

BEGIN: To be revised — Rough draft only!

Restoks are resource tokens; they represent rights to use certain resources according to system-defined policies. ***Detail later...***

Each Iguana object (memory section, thread, PD, EAS) has an *owner* PD, which is the PD whose creation method was invoked to create the new object. An object's owner is the entity whose restoks are charged for the object. Ownership does *not* imply any access privileges and is therefore not represented by capabilities.

How are restoks presented?

END: To be revised — Rough draft only

External address spaces are provided for support of legacy applications and applications too big to share the Iguana AS. An external address space (EAS) is the equivalent of a Linux process.

External address spaces operate to a restricted API (mostly raw L4 system calls) with little access to Iguana services. They are as such not well integrated into the Iguana system, and are in fact firewalled off the rest of the system in most respects. External address spaces may be used, for example, to support native Linux applications in full binary compatibility mode.

A thread running in an EAS cannot directly access memory in another EAS or the IAS. In order to access other memory it needs its owner to map in a region of the IAS (using L4 mapping IPC). The server may also map such a region to another EAS in order to facilitate inter-EAS sharing. An EAS can be single- or multi-threaded or (initially) be without any threads.

An EAS cannot present any capabilities to the system. Hence, threads running in an EAS are operating to a restricted system API. They can only perform operations which are allowed without capability presentation, such as L4 IPC to the owners. Other operations (like creating another EAS) need to be done on their behalf by their owner. For example in Linux emulation mode, a Linux process (running in an EAS) would execute the Linux *fork()* API, which is implemented as an IPC to a Linux server, which would create an EAS and return the child's PID to the caller.

With proper IPC control it might be possible to provide more access to Iguana services, however, we are highly doubtful that this would be a good idea, as EASen cannot be cleanly integrated into the Iguana access control model.

Hardware support is provided in the form of mechanisms for creating mappings of specific memory regions, handling DMA, and associating interrupt numbers with threads.

Open issue: There is no exception model defined yet for Iguana.

Chapter 3

Iguana API

Here we present the Iguana high-level API and provide examples of its use in a somewhat abstract, language-independent form. C language bindings are presented in Appendix E.

Information on failure modes of the various functions is discussed in the binding documentation in Appendix E.

3.1 Clients, servers, and objects

Iguana provides a client-server model of component interaction which is implemented using the underlying L4 IPC primitive.

On top of client-server invocations, Iguana implements a component model, where methods are invoked on instances. Component object invocations are performed by a server thread registered for a given *memory section*. The advantage of this model over a thread-per-object model is that references to objects are simply pointers to component instance data. This provides performance advantages for SMP systems, because multiple threads can be used to provide a given service. Access to these component objects is mediated through the use of capabilities, discussed below.

3.2 Capabilities

Capabilities define access rights on objects. When an object is created, the caller receives a *master capability* for the object. A *master* capability gives the holder maximum rights over the object, including the right to invoke any method, including methods which have not been registered at the time the *master* capability was created.

All objects provide *invoke* capabilities for the methods they offer. Memory sections have additional capabilities which are not associated with method invocations but with normal memory access operations: *read*, *write* and *execute* (often denoted as *R*, *W* and *X*, respectively). These convey the right to perform load, store or instruction-fetch operations on the memory section. These operations are logically considered method invocations, but no actual invocation occurs. In fact, it is not possible to invoke a method corresponding to these rights. Their whole purpose is to tie the normal memory operations into the same protection model. Read, write and execute are therefore called *pseudo methods*.

A further pseudo method is *Clist* (or *C*). This represents the right to insert a memory section as a Clist into a protection domain using the *insert* or *new_pd* methods (see Sections 3.3 and 4.1.2 for details).

When we talk about invocation rights below we include the pseudo methods, except where these are explicitly excluded.

Methods are grouped into *interfaces*. All methods belonging to the same interface share the same accessibility. Consequently, capabilities actually refer to interfaces rather than methods. The mapping from methods to interfaces is defined in an *interface definition*. Interface definitions are expressed in Iguana's *interface definition language* (IDL), discussed in Appendix G.

A capability is a data structure which contains (at least) an *interface ID* (IID). The IID consists of two parts: an *object ID* and an *interface number*. Interface number zero is never used, instead a capability with an interface number of zero is, by definition, a master capability. The IID has the form (size) of a memory address, but does not refer to an actual memory location. The actual breakdown (in terms of number of bits) between OID and interface number depends on the kind of object to which the capability refers.

There is one method that exist for all objects: the destructor. It removes the object from the system and renders all its capabilities useless:

```
obj->delete();
```

Each kind of object has its own set of standard methods which are available for all objects of that particular kind. These are discussed below.

Memory sections may, in addition to the standard methods available for all memory sections, have user-defined methods. A user-defined method is available only for the particular memory section for which it has been registered with the system.

No matter whether a method is user-defined, a standard method or the *delete* method available for all objects, it can only be invoked by a thread which holds an appropriate capability.

3.3 Protection Domains

Iguana's protection system is a capability-based system. In such a system a protection domain is said to be the union of all the capabilities held by the protection domain. In order to gain access to some of Iguana's resources, a protection domain must have the correct capability to access the required service. Iguana removes the need to specify the capability when requesting a system service meaning that all the security checks are done implicitly by the Iguana system.

Since protection domains are capabilities it is through capabilities that we can manipulate the protection domains. A user can either delete or add certain capabilities to whole Clists. Or if a finer detail is required you can then use the *add_clist*, and *remove_clist* methods.

Explicit L4 IPC is not to be used for communication between PDs (this is to enforce some PD encapsulation and allow communication through measureable sources - for resource management), only within a PD or between an external address space and its owner (see Section 3.8). Iguana method invocation, sessions and shared memory sections are the only legal communication mechanisms between PDs.

Implementation note: This restriction is presently not enforced. Refer to Section D.1.

create_pd creates a new PD owned by the PD on which this method is invoked.

```
pd_cap = pd->create_pd(flags);
```

The resources of the new PD are charged against the *owner*. The owner has no control over the created PD, unless it holds a capability to it. The new PD needs to be given a Clist in order to be able to execute any threads.

BEGIN: To be revised — Rough draft only!

If the owner's restok is destroyed (e.g. because the owner itself is destroyed) the owned PD is destroyed as well. This implies that all PDs owned by the destroyed PD are also destroyed.

***Will need restok arg to** create_pd*

END: To be revised — Rough draft only

Implementation note: The *flags* argument is used to specify whether IPC restrictions are enforced on this PD. It is not currently implemented. Refer to Section D.1.

delete deletes the PD.

```
pd->delete();
```

All resources allocated to the PD are released, including those by owned PDs are implicitly destroyed by this operation.

add_clist inserts a new Clist into the PD:

```
slot = pd->add_clist(clist);
```

The *clist* argument is a memory pointer, the caller must hold a valid *C* capability to the corresponding memory section. On success this will return the slot position where the Clist was stored.

Implementation note: The counterpoint to *add_clist*, *remove_clist*, is not currently implemented. Refer to Section D.3.

set_callback associates a callback buffer with this protection domain. Callbacks, which are implemented as circular buffers, allow for asynchronous communication: clients place their request in the buffer and it is handled when the server next checks the buffer, while the client continues to operate.

```
pd->set_callback(callback_buffer);
```

release_clist removes the Clist from the specified slot in the PD.

```
pd->release_clist(clist, pos);
```

The Iguana protection model is explained in more depth in Chapter 4.

3.4 Threads

Iguana threads are primarily L4 threads and can be manipulated by regular L4 system calls, such as *ExchangeRegisters*. However, certain operations on threads are privileged (in L4) and need to be performed by Iguana. This makes those operations subject to Iguana's protection model.

A complication is that Iguana's protection model requires the use of thread IDs that are different from L4's thread ID. Methods exist for mapping between the two IDs, and each method expects either an Iguana or an L4 TID.

Iguana provides the following methods for thread manipulation.

create_thread creates a new thread in a specified PD. This returns a capability to the thread object, as well as an L4 global thread ID (L4TID):

```
thread_cap = pd->create_thread([priority], &l4tid);
```

Creating threads in an EAS is discussed in Section 3.8.

The thread is created *inactive*. In the case of a local thread (created in the caller's own PD) it can be activated using L4's *ExchangeRegisters()* system call. If created in a different PD it can also be activated using *ExchangeRegisters()*, provided there is another thread in that PD which is told about the L4 thread ID of the new thread. Obviously this is impossible if the new thread is the first one in the PD.

The thread priority is an optional argument, representing a thread priority between 1 (lowest) and 255 (highest). If it is not specified, the default priority of 100 is used.

Implementation note: Iguana is not heavily reliant on L4 global thread Ids. Refer to Section D.2.

start starts (activates) an inactive thread. This is needed for activating threads in other PDs, as local threads can be activated via direct L4 system calls (but the method can be used on local threads as well).

```
tid->start(ip, sp);
```

The caller supplies the start address (*ip*) and initial stack pointer (*sp*) for the thread. The caller does not need any rights to the memory sections containing *ip* and *sp*, but the starting thread's PD needs them (*X* for the memory segment pointed to by *ip* and *RW* for the memory segment pointed to by *sp*). This method can only be called on inactive threads.

l4id is used to obtain the (global) L4TID of a thread, for passing to L4 syscalls such as ExchangeRegisters().

```
l4tid = thread->l4id();
```

id is the inverse of l4id and returns the Iguana thread reference given an L4 global thread ID.

```
threadid = l4tid->id();
```

domain returns the PD of a thread.

```
pd = thread->domain();
```

Implementation note: Currently not implemented. Refer to Section D.7.

myself is a static method that returns the caller's (Iguana) TID.

```
tid = myself();
```

delete deletes an Iguana thread.

```
thread->delete();
```

3.5 Memory sections

Memory sections represent virtual memory for data and instruction storage. Furthermore, memory sections can have application-defined methods implementing arbitrary functionality. This is the basic mechanism for the provision of services in Iguana: In order to provide a service, a memory section is associated with a server thread and a set of methods which are used to invoke the service. Method invocations result in a communication with the server thread.

The object ID of a memory segment is the number of its first page. Consequently, the IID for interface number zero (representing the *master* capability) is the first address of the memory section. This is referred to as the memory section's *base address*.

The standard methods available for all memory sections are as follows.

create_memsection is used to allocate a new memory section of a specified size in the current PD. The method returns the *master* capability for the new memory section (which is the object's base address).

```
memsection_cap = pd->create_memsection(size, &base);
```

delete removes the memory section and renders all its capabilities useless.

```
memsection->delete();
```

register_server registers a server thread for a memory section, replacing any server that may have been registered for the memory section previously. Methods may only be invoked if a server has been registered.

For discussion on how the memory section server dispatches method invocations see Section 6.1.

Implementation note: The return value is undefined. Refer to Section D.8.

```
memsection->register_server(server);
```

lookup returns the memsection and server thread associated with an object. This is required for session creation.

```
memsection = section->lookup(object, &server);
```

base returns the base address of a given memory section.

```
baseptr = section->base();
```

read, *write*, *execute* are pseudo-methods which are not directly invocable but only exist for their capabilities. Load, store, or instruction fetch require *read*, *write* or *execute* capability on the corresponding memory section.

BEGIN: To be revised — Rough draft only!

new_cap creates a capability for a specified interface number of an object. The caller must hold the *master* capability to the memory section.

```
cap = iid->new_cap();
```

Note that this may be used to create several different *invoke* capabilities for the same interface, including *read*, *write*, or *execute* capabilities. It can also be used to create additional *master* capabilities, although this probably doesn't make a lot of sense.

Capabilities for method invocation can be created whether or not a server is registered for the memory section. Invocation of a method can only work if a server has been registered.

validate checks whether a capability list grants *invoke* right to a certain interface.

```
if ( cap_list->validate(iid) ) { ... }
```

What sort of object is a* cap_list*? How does it get the* validate *method?

pin_range allows to pin a memory buffer to enable DMA-based I/O. The range is specified by a *start* and *end* address. The range must be wholly contained within a single memory section, and the caller must hold *read* and *write* capability to that memory section. The method returns a scatter-gather list, i.e. a list of physical frames that are to be used by the device driver.

```
sg_list = base_adr->pin_range(from,to);
```

Requires ??? capability on the memory segment plus ??? restoks.

unpin_range removes a previously established pinning.

```
base_adr->unpin_range(from,to);
```

FIXME: Memory pinning / unpinning not implemented yet.

END: To be revised — Rough draft only

BEGIN: To be revised — Rough draft only!

This model obviously requires a way for a callee to discover which thread implements any particular object, and a way for a server thread to be associated with a particular object. When creating an object the server registers itself by calling the *memsection_register_server* method on the memsection in which an object resides. As an example the timer server provides access to individual timer objects. On startup it registers itself as the server for the memsection it uses to allocate individual timer objects. On the client side, when given a capability to some object it will first call *session_create*, which will return the server thread to call. (On a security-enhanced L4 kernel it will also call the underlying L4 calls to allow the callee thread to perform IPC with the server thread.)

The described model works well when dealing with calling method on individual instances, for example a specific protection domain object, or timer object, however not all functionality involves calling methods on instances. Some servers need to provide some static methods, which are not associated with any particular instance. The most common example is a method used to create a new instance. For example the *timer_server_create* method creates new timer objects. In these cases it becomes less clear as to how the methods fit into the model. For these cases we treat the loaded server program as an instance in its own right. The current implementation treats the loaded program image as the instance for what would otherwise be static methods. On startup the program loader (see *iguana/init/src/init.c:start_server()*) will register the started thread as the server. To provide the ability to share text between two copies of the same server, the address used to refer to the server is that start of its data section.

END: To be revised — Rough draft only

3.6 Sessions

A session represents a communication channel between a client object and server thread.

There is an implicit session between all non EAS threads and the Iguana server thread.

Before invoking methods on an object a session must be established. A session is set up by calling the *session_create* method on the object we want to create the session with. After this the iguana server will then run some security checks on the client and server and check that the client thread has the right to call the server. The security checks involve checking that the client has the correct capability to communicate with the server.

BEGIN: To be revised — Rough draft only!

When set up, the session allows synchronous communication between the client and server. Calling the *add_async()* method requires a pair of ring buffers (one for calls, one for returns), which allows the client and server to communicate asynchronously, (without adding buffering to the kernel). When the *add_async()* method is called, the Iguana server makes another async upcall to the server thread to inform it of the new async buffer that has been set up.

This setup allows a client to set up an async session with a server without ever having to trust the server. ***(Compared to, for example, using a synchronous call to the server to set up the shared buffer, which would need a blocking call to the server.)***

END: To be revised — Rough draft only

When a session is deleted the threads can no longer communicate.

Implementation note: Iguana supports asynchronous notifications between sessions, but the API is currently subject to change and not documented. Refer to Section D.4.

Implementation note: Presently Iguana does **not** enforce the restriction that all inter-PD communication must be via sessions rather than raw L4 IPC. Refer to Section D.1.

create_session is a method on PD objects which creates a new session owned by the PD.

```
session_cap = pd->create_session(object, clist, server_thread);
```

The new session is established between the *object* and the *server_thread*. The resources of the new session are charged against the *object*. On creation a *session_created* call will be invoked on *object* and *server_thread*.

The *clist* parameter is optional. If omitted, the session is created with an empty clist.

delete deletes an existing session.

```
session->delete();
```

The session is deleted and both participants are informed. When the session is deleted the two PDs involved can no longer communicate.

BEGIN: To be revised — Rough draft only!

provide_access adds the capability to call the supplied object with the specified interface ID to the session clist.

```
session->provide_access(object, interface);
```

session_created is called when a session is established.

```
session_created(pd);
```

Called by iguana when a new session is established. This allows the PD to know that it is communicating with a new client and establish any resources required.

FIXME: *session_created* ***doesn't appear to be implemented***

session_deleted ... ***FIXME!***

add_async_buffer ... ***FIXME!***

buffer_added ***FIXME!***

new_session ***FIXME!***

add_async adds asynchronous communication buffers to the session.

```
session->add_async(call_buf, return_buf);
```

END: To be revised — Rough draft only

3.7 Resource Tokens

BEGIN: To be revised — Rough draft only!

The Iguana resource management model is explained in more depth in Chapter 5. Here we just provide a brief description of the API. This is unlikely to make a lot of sense at first reading, but is provided here for completeness. The reader is advised to skip details provided in this section and return here after reading Chapter 5.

set_restok sets the target PD's restoks, taking them from a specified PD's restoks:

```
target_pd->set_restok(from_pd, resspec);
```

Transfers fixed resource entitlements and establishes resource entitlement rates in *target_pd* as specified by *resspec*. The resources are taken from the restoks of *from_pd* (or refunded there if the new allocation is less than the previous value).

get_restok obtains the present restok balances and rates of a PD:

```
pd->get_restok(&resspec);
```

Restoks have a mask indicating type of resource they can be used for, presently thread, PD, EAS, VM, PM, time. Combination of refundable value and rent. Can create sub-restok, delegates part of refundable value/income stream, when destroyed refund value and stop income. When destroyed, charged objects are destroyed. Implicitly destroyed when owning PD is destroyed. Initially implement pure quota system (only refundable charges, no income).

Needs clarification and some more thinking.

END: To be revised — Rough draft only

3.8 External Address Spaces

External address spaces can be created with a PD (not an EAS) as the owner. External address spaces have their own thread-creation API, because they represent raw L4 address spaces. New EAS threads can, however, be manipulated using the regular Iguana thread API after creation.

An EAS cannot invoke Iguana methods, but must request services from its owner PD via L4 IPC. In particular, the owner is responsible for populating the external address space using L4 mapping IPC.

create_eas creates a new EAS owned by the PD.

```
eas_cap = pd->create_eas(kip, utcb);
```

The *kip* and *utcb* parameters specify the location of the L4 *kernel info page* and *user-level thread control block* in the new address space.

create_thread creates a new thread in an EAS:

```
thread_cap = eas->create_thread(pager, scheduler, utcb);
```

Here *pager* and *scheduler* are the L4TIDs of the page fault handler and scheduler of the new thread and *utcb* is the location of the new thread's user-level TCB.

delete deletes the EAS.

```
        eas->delete();
```

All threads associated with the EAS are deleted, followed by the EAS itself.

3.9 Hardware

The static *hardware* object is used for dealing with properties of physical hardware. The methods that can be called on this object are as follows.

back_memsection maps specific physical memory to a virtual memory section. This is used by device drivers to map device memory which can then be used for memory-mapped I/O via the memory segment.

```
        hardware->back_memsection(memsection, p_addr, attributes);
```

memsection is the memory object to back, *p_addr* a (suitably aligned) physical memory address, and *attributes* specifies any specific attributes for this backing, such as cache behaviour. On architectures that do not support memory-mapped I/O, *p_addr* refers to an I/O-space address.

Implementation note: The *attributes* flag may in the future contain architecture-specific flags, but currently none is implemented. Refer to Section D.5.

Implementation note: Currently there is no enforcement of access controls for this operation. Refer to Section D.6.

register_interrupt registers a handler thread, identified by an L4 TID, to receive a specific interrupt.

```
        hardware->register_interrupt(l4tid, irq);
```

3.10 Exceptions

Open issue: Iguana's exception model is not defined.

3.11 Synchronisation

Iguana does not provide a synchronisation service. A semaphore server for concurrency control on long critical sections can be implemented separately if required.

The approach Iguana uses for short critical sections is a notify on preemption system where by the user will turn the notify on before the critical section and turn it off after the critical section. What this means is that if during the critical section execution it gets preempted, the system will then jump back to the start of the critical section after the new critical section completes.

```
   NotifyOnPreemption();
   /* critical section */
   DisablePreemptionNotify();
   if (PremptionPending()) {
     Yield();
   }
```

Such a short critical section can be used to obtain or release a lock or to change a thread priority.

3.12 API Summary

Object	Method	Arguments	Return value	Section
pd	→*create_memsection*	(*size, *base*)	↦ *memsection_cap*	3.5
pd	→*create_pd*	(*flags*)	↦ *pd_cap*	3.3
pd	→*create_thread*	(*[priority], *l4tid*)	↦ *thread_cap*	3.4
pd	→*create_eas*	(*kip, utcb*)	↦ *eas_cap*	3.8
pd	→*create_session*	(*object, clist, server_thread*)	↦ *session_cap*	3.6
pd	→*set_callback*	(*callback_buffer*)		3.3
pd	→*add_clist*	(*clist*)		3.3
pd	→*delete*	()		3.3
session	→*provide_access*	(*object, interface*)	↦ *bool*	3.6
session	→*delete*	()		3.6
adr	→*register_server*	(*server*)	↦ *int*	3.5
adr	→*lookup*	(*object, &server*)	↦ *memsection_ref_t*	3.5
adr	→*base*	()	↦ *void **	3.5
tid	→*start*	(*ip, sp*)		3.4
tid	→*l4id*	()	↦ *l4tid*	3.4
tid	→*id*	()	↦ *thread_ref_t*	3.4
tid	→*domain*	()	↦ *pd*	3.4
tid	→*delete*	()		3.4
eas	→*create_thread*	(*pager, scheduler, utcb*)	↦ *thread_cap*	3.8
eas	→*delete*	()		3.8
hardware	→*back_memsection*	(*memsection, p_addr, attributes*)		3.9
hardware	→*register_interrupt*	(*l4tid, interrupt*)		3.9
	→*myself*	()	↦ *tid*	3.4

Memory sections also have the *read*, *write*, *execute* and *clist* pseudo methods. In the above table we use the following identifiers:

{pd,memsection,thread,eas,session}_cap: capability referring to an object of the respective type;

pd, *adr*, *tid*, *eas*: PD, Iguana thread, EAS reference (object ID part of the object capability).

Should an arbitrary IID of the object be allowed too? Probably yes. In fact, for memory sections we want to allow an arbitrary address within.

Restoks need further consideration.

hardware: the static object representing operations on hardware;

priority, *attributes*, *interrupt*, *p_adr*, *flags*, *interface*: integers;

kip, *utcb*, *ip*, *sp*, *base*: virtual memory addresses;

callback_buffer: memory section reference;

object: object reference;

l4tid, *pager*, *scheduler*: L4 thread IDs;

server_thread: Iguana thread IDs;

clist: special data structures.

Chapter 4

Protection Management

Iguana features a general and flexible capability-based protection system which is able to emulate a number of standard access-control policies.

4.1 Capabilities and Protection Domains

A capability [**?**] is an unforgeable token that is *prima facie* evidence of some right the holder possesses.

In a capability system a thread's protection domain (i.e., the sum of its access rights) is equivalent to a set of capabilities — the same is true for Iguana. What distinguishes different capability systems from each other are the representation of individual capabilities and the representation of protection domains (i.e., the precise way in which a protection domain is defined as a set of capabilities).

Capabilities can be stored as kernel-owned and -maintained capability lists (leading to a *segregated* capability system) or as user data. In the latter case the capabilities must be protected from forgery. This can either be done by hardware means (*tagged* capabilities) or sparsity (*sparse* capabilities). Iguana uses the latter approach, as explained in Section 4.1.1.

In a capability system an appropriate capability must be presented to the system whenever a system service is obtained. This presentation can be *explicit*, meaning the capability (or a reference to it) is passed as an explicit argument to a method invocation. The alternative is *implicit* presentation of capabilities, which separates protection from function and is thus less intrusive (similar to systems using protection based on *access-control lists*). Iguana uses implicit presentation via two-level data structures called Clists. These are explained in Section 4.1.2.

4.1.1 Iguana capabilities

Iguana capabilities are user-level objects, in the sense that they are data structures which can be read and written like any other data by unprivileged threads.

As explained in Section 3.2, an Iguana capability is a data structure which contains an interface ID, which itself is composed of an object ID and an interface number. Section 3.2 did not explain what other information a capability contains. This is because for most purposes the remainder of a capability is opaque data.

Iguana capabilities are implemented as *password capabilities* [**?**], meaning that the balance of the capability's data is simply a random bit-string — a password. The password protects a capability from forgery, as, in order to manufacture a valid capability, one needs the correct password that matches the IID.

Iguana maintains a system-wide list of valid capabilities; whenever a new capability is created (via one of the constructors *new_mem*, *new_thread*, *new_pd*, *new_eas* or *new_cap*), Iguana enters it in its internal data structures before returning it to the caller. When verifying a capability, Iguana looks for a match with the internal data structures; only if it is found there is the capability considered valid. The size of the password is a system generation parameter so it can be adapted to the system's security requirements.

Since capabilities are regular data, they can be passed around freely. Any threads which can communicate can pass capabilities to each others. In this way, it is possible to pass access rights to data between threads without explicit system interaction.

However, whether a thread which receives a capability can make any use of it is a different matter, which has to do with the way protection domains are defined in Iguana. By controlling the data structures which define protection domains, it is possible to limit the propagation of access rights even between communicating threads. How this is achieved is explained in the following sections.

4.1.2 Capability lists

As mentioned above, Iguana's password capabilities are stored in user-level capability data structures called *Clists*. They are implicitly presented to the system on a method invocation, meaning that the system knows the location of the caller's Clists and performs a lookup on them as required.

User-level capabilities are convenient to use, but often make it impossible to enforce system-wide access-control policies. In order to to enable enforcement of such policies, Iguana's Clists contain a level of indirection which allows the interposition of security policy managers. This is explained in more detail in Section 4.2.2.

In order to provide this level of indirection, Iguana's capability storage uses a two-level data structure. The system represents the protection information of a protection domain as an array of *Clist capabilities*. Each of these is a capability (conferring the *C* right) to a memory section which is interpreted as a Clist (a system-defined data format). A new protection domain is created empty, i.e., with no Clists. ***FIXME!*** The creator (or anyone who obtains the appropriate capabilities from the creator) then uses system calls to add Clists to the new PD. The system then validates that the caller possesses at least the *C* right on each of these Clists before inserting them into its internal representation of the PD.

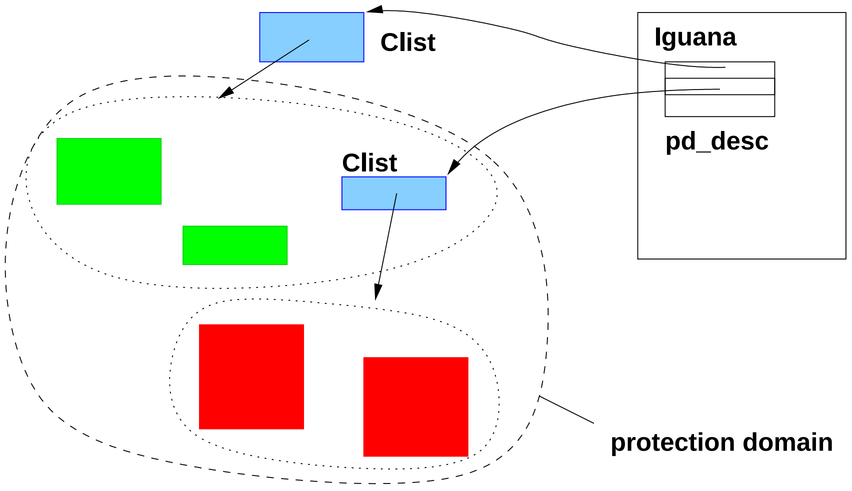

Figure 4.1: Clists defining a protection domain

This two-level scheme is depicted in Figure 4.1. Iguana's data structures describing a PD contain an array of Clist pointers, the entries of which are called *slots*. Each of the Clists referenced by the pointers in those slots contains a set of capabilities, corresponding to access rights for a set of objects; such a set comprises a sub-PD. The union of the sub-PDs defined by the Clists defines the protection domain (at least as far as access rights go).

When validating a method invocation, Iguana searches the PD's Clists for a capability that matches one of the valid capabilities for the method. Validations are cached for efficiency.

The method of searching within a Clists depends on the specific Clist format. Two formats are supported: a *sorted* (by ascending IID) and an *unsorted* one. Binary search is used on the former, a linear scan on the latter. The search is terminated when a capability for the invoked interface is found. Any invalid capabilities encountered are quietly ignored. It is obvious that Clists should be kept in sorted format whenever possible.

Implementation note: Presently only the unsorted format is implemented. Refer to Section D.9.

4.2 Managing Protection Domains

The specific representation of protection domains used in Iguana has a number of interesting properties:

1. capabilities are user-level objects. This means they can be passed around freely without system intervention;
2. capability presentation is implicit: no method invocation takes an explicit capability argument. This makes the protection system unintrusive;
3. a protection domain may or may not hold the capabilities to the Clists which define it. This makes it possible to set up PDs with access to certain objects, without giving the PD any access to the capabilities to those objects. Such PD is not able to propagate its access rights to others.

For example, in Figure 4.1 the first Clist is outside the protection domain it helps to define. Hence the PD has no access to the capabilities for the objects in the corresponding sub-PD. The capabilities of the second sub-PD are accessible to the PD (assuming the rights granted to the second Clist by the first sub-PD include R).

Together these properties provide a great amount of flexibility in tailoring protection domains and controlling their interaction. As will be discussed in more detail in Section 4.2.2, a number of standard security models and policies can be implemented using this scheme.

4.2.1 Manipulating protection domains

According to what was said above, there are two ways of manipulating protection domains: manipulating the Clists objects (e.g. adding or removing individual capabilities to a Clist) or manipulating the Clist array (inserting or removing whole Clists). The former requires W access to the Clist objects, while the latter requires invocation rights to the *add_clist* and *remove_clist* methods.

Manipulating Clists provides fine-grained control over PDs. A thread which creates a new object will typically add the new object's capability into one if its Clists in order to be able to invoke methods on that object. It may also pass the capability (or specific interface capabilities) to threads in other protection domains, which will then insert it into one of their Clists. This is the basic way of sharing objects in Iguana.

Adding or removing Clists performs coarse-grain control over PDs. It is typically used for grouping and sharing capabilities for related objects. This is akin to using group access rights in Unix systems.

However, Clists can also be used for sharing objects without handing out actual obje‘ct capabilities. By putting a set of capabilities into a separate Clist, and handing out a C capability to that Clist, other PDs can use the objects without being able to access (and distribute) the actual object capabilities themselves.

Implementation note: Since the current L4 kernel does not have an efficient IPC control mechanism we use a server side validation protocol to make sure that a client has the correct Clists for the operation they want to perform. Refer to Section D.10.

4.2.2 Implementing various protection models

The indirection of Iguana's Clist structure can be used to confine untrusted code. In order to run an untrusted program securely, the caller sets up a protection domain which contains no Clist capabilities. The caller also ensures that the untrusted PD does not contain *W* capabilities, except for buffers used for communication with the caller. The caller does not hand *W* capabilities to those buffers to any other PDs. If the untrusted code is then executed in the untrusted PD, it cannot leak any data, even if it allocates new memory sections or its code contains embedded *W* capabilities. An example of this is shown in Figure 4.2.

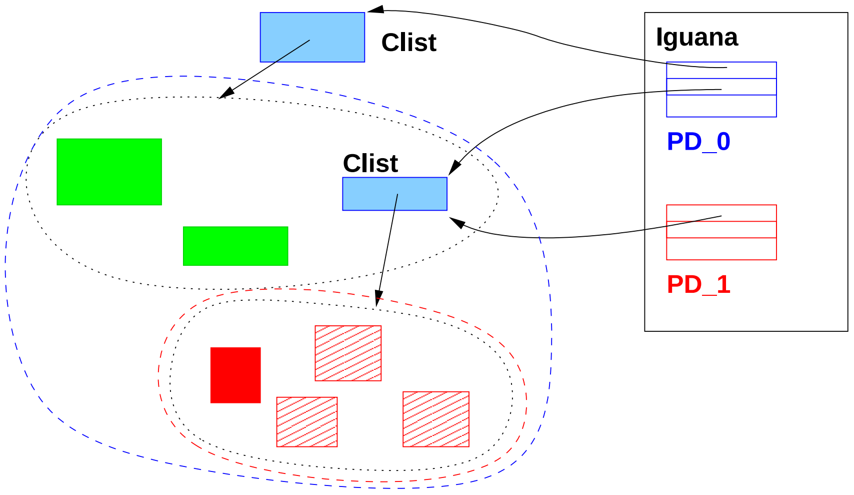

Figure 4.2: Encapsulation of a protection domain (PD_1). Shaded boxes represent read-only or execute-only objects. If no third PD has a capability to PD_1's only writable object, then PD_1 cannot pass data to anyone but PD_0

Alternatively, the hierarchy can be used to simulate a segregated capability model. In this case, one (ore more) protection server "owns" all Clists (in the sense that only it holds Clist capabilities). These servers also must be the sole holders of PD-creation rights, so all *create_pd* operations must be performed by the protection server on behalf of a client.

In such a scenario, the protection server is the sole authority which decides what access rights any PD has, and how access rights can be transferred. The server can implement standard models of mandatory access control, such as Bell-LaPadula [**?**] or Chinese Wall [**?**]. Figure 4.3 shows an example.

Can this do DTE/RBAC? DTE = domain type enforcement RBAC = role based access control

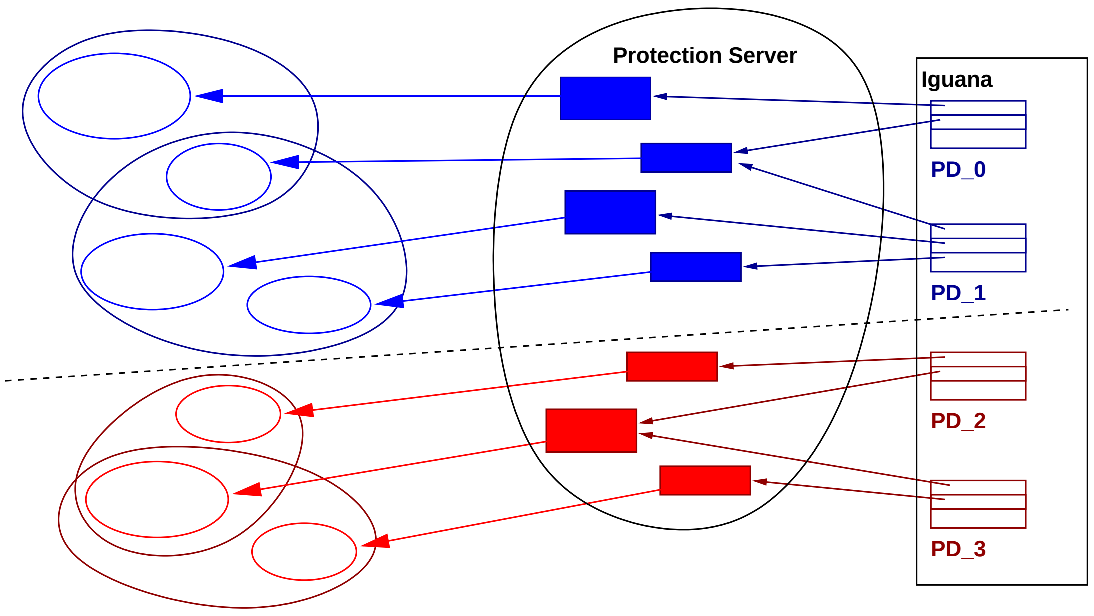

Figure 4.3: Capability segregation using a protection server. In this example, the server enforces an isolation policy between the red and blue protection domains

Chapter 5

Resource Management

BEGIN: To be revised — Rough draft only!

None of this is presently implemented.

Iguana manages resources using an economic model [**?**]. The present model is a generalisation of the *bank account* model used by Mungi for charging for disk usage [**?**]. Iguana's resource management model combines Mungi's rent model with the ability to define simple quota.

The basic idea is that each resource has both a *value* and a *rate*; either or both of which may be zero. The value is the amount of currency that must be paid by the (prospective) owner who wants to allocate the resource. The rate is the amount that must be paid per unit of time ***what time?*** during which the resource is held. The resource's value is refunded to the owner when the resource is deallocated, while the rate is non-refundable.

5.1 Resource charging

The current value and rate of each resource is determined system-wide by the *resource manager*, which is an Iguana server. Each resource's value and rate is set by the resource manager independent of other resources. Presently the following resources exist:

- virtual memory
- physical memory (???)
- protection domains
- external address spaces
- sessions
- threads
- processor time.

The resource manager may choose to have a zero rate and a non-zero value for a particular resource; this corresponds to paying a purchase price at allocation which is refunded at deallocation. Alternatively, the resource manager can choose to set the value of a resource to zero and the rate non-zero; this corresponds to paying (non-refundable) rent for the resource. If the value and rate are both non-zero, then the resource requires both a purchase price and rent payments. Of course, the resource manager can choose to set a resource's value and rate both to zero, which turns off management of that resource, limiting its use only by system-wide availability.

The resource manager may choose to keep a resource's value and rate constant or may vary them over time. For example, the value or rate of a resource may be increased in response to high utilisation, in order to encourage clients to return unneeded resources. If a resource is deallocated, its *current value* is refunded to the owner, not the original value at the time of allocation. ***FIXME: This could lead to hoarding/speculation, we may need to rethink this.***

The rate is charged to clients periodically in advance. At the time a resource is allocated, the value and rate for the initial charging period is collected from the client's resource tokens. Allocation fails if the client has insufficient restoks.

What happens when there are insufficient tokens for paying for rates? Owner blocked or killed? Or some resources deallocated? Which? Do we need a prioritised victim list?

Rent collection may happen prematurely (when we have a resource shortage). In this case excess rent is refunded prior to charging the next rent at the (presumably) higher rate.

Need an API for communicating with resource manager. At lease need to find out present values, rates and charging period.

5.2 Resource accounts

In order to pay for resource use, each protection domain has a set of *resource accounts*, one for each resource type. Like the corresponding resource, each account has a value as well as a rate. Whenever a PD allocates some resource, the resource's value is subtracted from the value of the (prospective) owner's resource account, while the resource's rate is subtracted from the ...

Open issue: Are restoks first class objects or are they only attributes of PDs?.

5.3 Income

Restoks are drained by rent, need income to offset. Paymaster deposits rate$\times$time periodically, as in [?].

5.4 Taxes

Rates must accumulate into capital, as otherwise rates behave exactly like values and are redundant. This implies that we also need taxation, as in [?].

Well-known taxation formula and public taxation rate.

5.5 Granting resource tokens

Can create new restoks. They get a value by transfer from an existing restok, they get a rate by drawing on an existing restok (either can be zero, of course). If restok is destroyed, remaining value reverts to source, rate ceases to draw on source.

5.6 Resource management models

There is a top-level restok which is the source of all restok rates and values. Presumably held by the resource manager. Resource manager can decide on the policy by setting value/rate of resources and the restoks it hands out. Zero rates on all restoks of a particular type implements a simple quota model. Zero value but non-zero rates and 100% taxation corresponds to a proportional share model.

We need to investigate how standard models for each resource type fits into this.

END: To be revised — Rough draft only

Chapter 6

Iguana Services

An Iguana service is composed of one or more threads which provide functionality to client threads via L4 IPC. Iguana services are optional, though some may depend on others. For example, several services register themselves with the naming service, which provides a flat namespace and is discussed in more detail below.

Services typically exist to arbitrate access to a shared resource, such as a device or a naming pool, or to provide functions which would otherwise require additional threads in the client, such as a timer service. These requirements distinguish services from regular library code.

Services are specified as part of a bootimage target. Use the bootimage target in `configs/iguana.sconf` if you want to compile a different set of services to the ones Iguana includes by default.

6.1 Memory section server

Like other services in the Iguana system the memory section server uses L4 IPC to talk to the server. Unlike other services however, the memory section server is built in to the iguana system as an integral part. Meaning that the memory section server is not built (and placed in the bootimage image) in to the system at compile time, but built into the Iguana image itself.

6.2 Establishing Sessions

Sessions are Iguana's way to regulate communication between protection domains. A session can also be used to enforce PD encapsulation. When creating a session Iguana will create a Clist for that session.

The Iguana system may then check to see if a session is already set up with the server, if it is the session creation protocol will halt. If no existing session is set up, then the Iguana server is contacted.

It is the job of the Iguana server to ensure that the session is valid between the two communicating PD's. Meaning that it will check the capabilities of the initiator and see whether it has the capability to establish as session with the receiving PD. If all the security checks are passed then the Iguana server will create a master capability for the session.

the process of establishing the actual session between the two communicating PD's is then quite simple. A session object is created with no callback buffers (as these can be added if needed at a later time). Then both the client and server PD session lists are updated. Finally a Clist gets added to the server PD.

6.3 Naming

The naming server is similar to other services where by it uses L4 IPC to communicate with the server. Like the others it uses the IDL to hide all of the complicated low level code required to communicate with the naming server thread.

Iguana's naming service is used by many other services throughout the system. This means that for most system configurations the naming service will be compiled in to the Iguana system.

Naming methods which are not involved in the client server mechanism of the service are stated below

create_name_node creates a name node in the naming information list.

do_notify explicitly sends an L4 IPC to the thread to notify.

notify_list iterates the notify list if there is a match will notify the matching thread.

6.4 Timer

One of Iguana's services it offers is the timer service. This service can be used to receive incoming timer requests from devices. Iguana's timer service offers the ability to insert, delete, allocate and deallocate items in the timer queue. Like all services the timer server will poll waiting for an incoming request. An incoming request can be from an interrupt, a device or the system clock. Upon receiving the request the server will jump to the corresponding function in relation to what was the calling method.

When a new timer gets created, the system will insert it in to the inactive timer list. What is returned from this is a capability to the timer event. Currently the system only returns a master capability but in the future, it will be possible to add any type of access to the newly created timer item.

The timer callback mechanism uses L4 IPC to communicate with the timers owner thread. This communication method passes the timer mask to the receiving thread. The callback will then iterate through the list of active timers to check whether or not the current timers timeout value is greater than the system clock. In the case where the timer is not greater than the real system time, the system will set up the timer to timeout at the specified time. If it is below the current time it will just continue to iterate through the list until it reaches a timer whose timeout value is above the current system time.

Timer methods which are not involved in the client server mechanism of the server are stated below

make_active takes an inactive timer item and removes it from the inactive timer list and places it in the correct posityion in the active list.

delete_inactive deletes an inactive timer from the timer list.

insert_inactive inserts an inactive timer into the timer list.

make_active makes an inactive timer active. It also removes the timer from the inactive list (and places it in the active list).

deactivate_timer makes an active timer inactive, also removes timer from active list, and places it in the inactive list.

Chapter 7

System Startup

7.1 Making a Boot Image

To boot an iguana system you need to load the L4 kernel, a σ_0, Iguana itself, and any initial services. We provide a tool that makes it simple to create a boot image from a set of normal ELF files.

We use a tool called `dite` to create a bootimage from a set of input files. The basic function of dite is that it creates a single ELF file, containing all the program sections of the input files. Secondly it creates a bootinfo structure (as per the L4::Pistachio reference manual) describing the input files. Finally it also patches the address of the bootinfo structure, σ_0 and root task into the kernel configuration page.

7.2 Iguana σ_0

This section is pretty unclear [GH]

In an L4 based system, σ_0 typically acts as the root task's pager. On startup σ_0 will acquire all free physical memory in the system (which it will map one to one with the virtual addresses of σ_0) and pass it to the root task when it is requested. This is also called an hierarchical paging system since the user-level application (in our case the root task) that will receive the pages from σ_0 will then allocate them on to other faulting tasks,, through the use of a virtual memory system (or some other system). The method L4 uses to complete the passing of memory from σ_0 to the root task is IPC, where the message contains an *fpage mapping* of a system flexpage (whose size is architecture-dependent, multiple sizes may be supported).

σ_0 is run as its own task, this means that during boot time it will need to collect its own information about the memory usage in the system. σ_0 obtains this information about the system through the use of the lower L4 kernel interface, more specifically the `L4_GetKernelInterface` function. As described in Section 7.1, L4 will know where in the boot image each part of the Iguana system is located during start up. Since all relevant information about σ_0 and the root task is stored in the L4 kernel configuration page, we can use normal L4 calls to access this information quickly and easily. An advantage of this method is that both σ_0 and the root task can access this information at the same time and work from the data gathered from the kernel interface page.

In order to process requests from only valid regions of memory, σ_0 will create a boot map memory image which stores all the valid regions of the root task, as well as all the valid memory regions from the L4 kernel configuration page. For example in a page-fault request, σ_0 will search through the boot map image to see if the faulting address is actually a valid address before it completes the mapping. The boot map gets its information about valid regions of memory by iterating through the memory descriptors from the previously acquired kernel configuration page.

σ_0 accepts pagefaults and memory requests from the root task only. As a result, σ_0 just sits in a loop waiting to process requests from the root server. A page fault occurs when the root task faults on an address. A page request will occur when the root task explicitly asks for an address to be mapped. When a page fault occurs, σ_0 maps a valid physical address stored in the boot map structure. When a page request occurs, we just map the requested address to a fpage and map. The difference between the two operations is that with a pagefault the mapped memory has its sendbase set as the faulting address, that is, the memory is mapped to a virtual address, whereas the pagerequest does not do this, it just maps an fpage of the address passed to it.

7.3 Booting an Iguana System

BEGIN: To be revised — Rough draft only!

At startup the first application passed is started and given all the caps to the system. It then chooses policy of what to run and with which caps.

In reality we have a script language "conf" (see my thesis) and a simple default initialisation which parses a provided conf file (which is just a file in the bootimage) and then starts the system based on that.

END: To be revised — Rough draft only

Chapter 8

Device Drivers

Low level drivers are programmed to the device driver framework model. Iguana defines the interfaces and model of interaction between device drivers. Device drivers interact via a shared command buffer mechanism. This allows efficient asynchronous communication between the client and driver.

An Iguana device driver consists of three parts

- A generic library interface which contains functions to setup and tear down direct memory access. This library also has the interrupt jump functions for use when an interrupt is called.
- A device specific library which is explicitly written for each device.
- A class specific server which gets compiled against the device specific part to produce a device driver server.

For the device specific part of the device driver, the device driver writer has to write methods to do the following

setup called to initialise device (interrupts are not enabled here).

```
static void *setup(int spacec, bus_space_t *spacev,
                   dma_handle_t dma, bus_space_t pciconf);
```

enable start the device (interrupts enabled now).

```
static void enable(void *device);
```

cleanup called to cleanup the device.

```
static void cleanup(void *device);
```

interrupt called when an interrupt is called.

```
static void interrupt(void *device);
```

All drivers will have a data structure which holds information about the drivers operations. These operations are different depending on what type of device is being used. For example, in the case of a serial driver the individual operations are read and write.

Each driver needs to have its own device driver server, built into the bootmap image during the boot image creation.

The device driver framework used in the Iguana system places all of the driver code at a user level. This should be obvious from the fact that each driver needs a driver server and is placed into the boot image at build time.

Each device will be represented in the system by a data structure which will be able to be cast to a driver instance. In this way we can allow an individual device to be represented as a driver in the system.

Each driver instance has an asynchronous callback buffer for reading and writing to the device, this asynchronous communication is detailed elsewhere in the manual (as well as the reasons behind it).

Appendix A

Kenge Library Summary

Appendix B

Build Instructions

BEGIN: To be revised — Rough draft only!

In order to build the Iguana system (and its related components) you will need to have the following tools installed.

tla is used for revision control. It can be found at http://gnuarch.org/

python 2.3 for use by the *tla*command and other related build tools. It can be found at http://www.python.org/2.3.4/

SCons is the actual build system. It can be found at http://www.scons.org/

toolchain to cross compile the source into a native binary. It can be found at http://kegel.com/crosstool/

dite for patching multiple binaries into a single binary for use by the bootloader

Once the tools are installed we can get on to builiding the system. It should be noted here that we will use the ARM architecture as an example.

1. First we need to get the sources. We can either download the release tarball from http://www.disy.cse.unsw.edu.au/Software/Iguana/ or use the following *tla* command to get the release.
   tla get disy@cse.unsw.edu.au–2004/iguana-project–releases iguana-project
   or for more recent code we can use
   tla get disy@cse.unsw.edu.au–2004/iguana-project–mainline iguana-project
2. Set up the sources. To do this we can use the command (from inside of the iguana-project directory)
   tla build-config iguana
   this will build the external parts of the Iguana system (for example wombat)
3. Build the Iguana system. Run the command
   scons
   arm-elf-run build/bootimg.dite

END: To be revised — Rough draft only

Appendix C

Commented Example

Give a longish listing of fragments of real code, with a line-by-line explanation next to it (or better in-line comments?)

BEGIN: To be revised — Rough draft only!

Eg: 1 - *process_create()* / *fexec()*

```
    // Setup new PD
    new_process = my_pd->new_pd();
    new_process->set_restok(my_pd, restok);
    // or should this be my_pd->transfer_restok(new_process, restok)?
    stack_c = new_process->new_mem(DEFAULT_STACKSIZE);
    data_c = new_process->new_mem(data_segment_size);
    clist_c = new_process->new_mem(DEFAULT_CLIST_SIZE);
    add_cap_to_clist(clist, stack_c);
    add_cap_to_clist(clist, clist_c);
    add_cap_to_clist(clist, data_segment_c);
    add_cap_to_clist(clist, new_process_c);

    main_c = new_process->new_thread(main_l4tid);
    main_c->start(entry_pt, stack_c.id);
```

END: To be revised — Rough draft only

Appendix D

Implementation Restrictions

D.1 PD encapsulation

The L4Ka::Pistachio kernel, on which Iguana is presently implemented, does not support an efficient mechanism for encapsulating protection domains (i.e., restricting a PD's communication to sessions). Flags for `create_pd` will be used to enforce this encapsulation (using L4 *redirectors*), but this will impose significant run-time overhead.

The `create_pd` function does not currently accept any flags, and hence encapsulation is not presently enforced. Instead we presently rely on a server-side protocol for ensuring that PDs only communicate via sessions to which they hold capabilities. For description of the server side protocol please see Section D.10.

The need for this protocol will be eliminated once `create_pd` flags are implemented (although the run-time cost will be higher than that of the server-side protocol).

This will all become redundant once a security-enhanced L4 kernel provides an efficient IPC control mechanism; this is expected to be available by mid 2005. Once this is in place, PDs will *always* be encapsulated, without the need for either server-side protocols or significant run-time overheads.

D.2 L4 global thread Ids

L4 is moving away from a global thread Id model. Iguana is not heavily tied to global thread Ids and will support the new model when it appears.

D.3 `remove_clist`

There is currently no way to remove a clist once it has been added.

D.4 Async communications for sessions

The current session async API, as viewable in Appendix E, is relatively recent and subject to change.

D.5 Attributes for `back_memsection`

Currently the only attributes supported are for cached and uncached backing, but in the future other attributes may be added, either globally or on a per-architecture basis.

D.6 Memory section rights

Currently no specific access rights are needed to use the *back_memsection* method. In future there may be some security checks on whether or not the user has *RWX* capabilities on the memsection.

D.7 Domain of a thread

Finding a protection domain of a thread is currently not implemented.

D.8 Register server return value

This method is defined as returning int, but the actual service is void. This means that the wrapper returns basically random stuff.

D.9 Unsorted Clists

Clists are currently only stored in an unsorted format. In the future they may be stored in sorted format, but this is still under discussion. The unsorted format mostly serves to avoid race conditions while updating Clists.

D.10 Server side protocol

As described previously Iguana does not have any in built protection mechanisms for ensuring PD only communicate through the session mechanism. As such a server side protocol is used to enforce this restriction.

On every call to the Iguana server a security check can be performed on the client to see if the client has the correct Clist to perform the required operation.

This security check first finds which memory section the object we want to access is in. It then iterates through the Clists of the protection domain to see if it has a matching capability. If a match occurs the operation the client wants to perform will proceed.

Appendix E

C Bindings for Library API

E.1 libs/iguana/include/iguana/types.h File Reference

```
#include <l4/types.h>
#include <stdint.h>
```

Classes

- struct **cap_t**

Defines

- #define **INVALID_CAP** { .ref.obj = 0 }
- #define **IS_VALID_CAP**(x) (x.ref.obj != 0)

Typedefs

- typedef uintptr_t **objref_t**
- typedef **objref_t memsection_ref_t**
- typedef **objref_t thread_ref_t**
- typedef **objref_t pd_ref_t**
- typedef **objref_t session_ref_t**
- typedef **objref_t eas_ref_t**
- typedef **objref_t hw_ref_t**

E.1.1 Define Documentation

E.1.1.1 #define INVALID_CAP { .ref.obj = 0 }

E.1.1.2 #define IS_VALID_CAP(x) (x.ref.obj != 0)

E.1.2 Typedef Documentation

E.1.2.1 typedef objref_t eas_ref_t

E.1.2.2 typedef objref_t hw_ref_t

E.1.2.3 typedef objref_t memsection_ref_t

E.1.2.4 typedef uintptr_t objref_t

E.1.2.5 typedef objref_t pd_ref_t

E.1.2.6 typedef objref_t session_ref_t

E.1.2.7 typedef objref_t thread_ref_t

E.2 libs/iguana/include/iguana/memsection.h File Reference

```
#include <stdint.h>
#include <l4/types.h>
#include <iguana/types.h>
```

Defines

- #define **MEM_NORMAL** 0x1
- #define **MEM_DIRECT** 0x2
- #define **MEM_DMA** 0x4

Functions

- **memsection_ref_t memsection_create** (uintptr_t size, uintptr_t ∗base)

  Create a new memory section.

- **memsection_ref_t memsection_create_fixed** (uintptr_t size, uintptr_t base)

  Create a new memory section at a fixed location.

- **cap_t _cap_memsection_create** (uintptr_t size, uintptr_t ∗base)
- int **memsection_register_server** (**memsection_ref_t** memsect, **thread_ref_t** server)

  Register a server thread for this memory section.

- **memsection_ref_t memsection_lookup** (**objref_t** object, **thread_ref_t** ∗server)

  Return the memsection and server thread for the supplied object.

- void ∗ **memsection_base** (**memsection_ref_t** memsect)

E.2.1 Define Documentation

E.2.1.1 #define MEM_DIRECT 0x2

Memory whose virtual address equals its physical address

E.2.1.2 #define MEM_DMA 0x4

DMA-able memory.

E.2.1.3 #define MEM_NORMAL 0x1

Normally-allocated virtual memory

E.2.2 Function Documentation

E.2.2.1 cap_t _cap_memsection_create (uintptr_t *size*, uintptr_t ∗ *base*)

E.2.2.2 void∗ memsection_base (memsection_ref_t *memsect*)

Return the base address of a given memory section

Parameters:

← ***memsect*** The memory section to lookup

Returns:

The base address. NULL on failure

E.2.2.3 memsection_ref_t memsection_create (uintptr_t *size*, uintptr_t ∗ *base*)

Create a new memory section.

Parameters:

← ***size*** The size of the memory section in bytes

→ ∗***base*** The (virtual) base address of the new memory

Returns:

A reference to the new memory section.

E.2.2.4 memsection_ref_t memsection_create_fixed (uintptr_t *size*, uintptr_t *base*)

Create a new memory section at a fixed location.

Parameters:

← ***size*** The size of the memory section in bytes

← ***base*** The virtual base address of the new memory

Returns:

A reference to the new memory section.

E.2.2.5 memsection_ref_t memsection_lookup (objref_t *object*, thread_ref_t $*$ *server*)

Return the memsection and server thread for the supplied object.

Parameters:

← ***object*** The object for which memsection and server information is required

→ $*$***server*** The server thread for the supplied object

Returns:

The memsection for the supplied object.

E.2.2.6 int memsection_register_server (memsection_ref_t *memsect*, thread_ref_t *server*)

Register a server thread for this memory section.

Parameters:

← ***memsect*** The memory section to register the server for

← ***server*** Iguana thread reference to the server thread

Returns:

Undefined.

E.3 libs/iguana/include/iguana/pd.h File Reference

```
#include <l4/types.h>
#include <iguana/types.h>
```

Functions

- **pd_ref_t pd_myself** (void)

  Return a reference to the current protection domain.

- **pd_ref_t pd_create** (void)

  Create a new PD owned by the current PD.

- **pd_ref_t pd_create_pd** (**pd_ref_t** pd)

  Create a new PD owned by the specified PD.

- void **pd_delete** (**pd_ref_t** pd)

  Delete protection domain.

- **thread_ref_t pd_create_thread** (**pd_ref_t** pd, L4_ThreadId_t $*$thrd)

  Create a new thread in the specified protection domain.

- **thread_ref_t pd_create_thread_with_priority** (**pd_ref_t** pd, int priority, L4_ThreadId_-t ∗thrd)

  Create a new thread in the specified protection domain with the specified priority.

- **memsection_ref_t pd_create_memsection** (**pd_ref_t** pd, uintptr_t size, uintptr_t ∗base)

  Create a new memory section in the specified protection domain.

- void **pd_set_callback** (**pd_ref_t** pd, **memsection_ref_t** callback)

  Setup an async communications channel for messages.

- uintptr_t **pd_add_clist** (**pd_ref_t** pd, **memsection_ref_t** clist)

  Add a clist to this protection domain.

E.3.1 Function Documentation

E.3.1.1 uintptr_t pd_add_clist (pd_ref_t *pd*, memsection_ref_t *clist*)

Add a clist to this protection domain.

Parameters:

← ***pd*** The protection domain to add the clist to

← ***clist*** The new clist to add

E.3.1.2 pd_ref_t pd_create (void)

Create a new PD owned by the current PD.

Returns:

A reference to the new PD

E.3.1.3 memsection_ref_t pd_create_memsection (pd_ref_t *pd*, uintptr_t *size*, uintptr_t ∗ *base*)

Create a new memory section in the specified protection domain.

Parameters:

← ***pd*** The containing protection domain

← ***size*** The size of the new memory section in bytes

→ ∗***base*** The base address of the new memory section

Returns:

A reference to the new memory section

E.3.1.4 pd_ref_t pd_create_pd (pd_ref_t *pd*)

Create a new PD owned by the specified PD.

Parameters:

$\leftarrow$ ***pd*** The owning PD

Returns:

A reference to the new PD

E.3.1.5 thread_ref_t pd_create_thread (pd_ref_t *pd*, L4_ThreadId_t $*$ *thrd*)

Create a new thread in the specified protection domain.

A new thread is created in the supplied PD. The L4 thread ID of the new thread is stored in the thrd param, and the Iguana thread ID is returned.

Parameters:

$\leftarrow$ ***pd*** The containing protection domain

$\rightarrow$ $*$***thrd*** The L4 thread ID of the new thread

Returns:

A reference to the new Iguana thread ID

E.3.1.6 thread_ref_t pd_create_thread_with_priority (pd_ref_t *pd*, int *priority*, L4_ThreadId_t $*$ *thrd*)

Create a new thread in the specified protection domain with the specified priority.

A new thread is created in the supplied PD. The L4 thread ID of the new thread is stored in the thrd param, and the Iguana thread ID is returned.

Parameters:

$\leftarrow$ ***pd*** The containing protection domain

$\leftarrow$ ***priority*** The thread's initial priority, from 1 to 255.

$\rightarrow$ $*$***thrd*** The L4 thread ID of the new thread

Returns:

A reference to the new Iguana thread ID

E.3.1.7 void pd_delete (pd_ref_t *pd*)

Delete protection domain.

Parameters:

$\leftarrow$ ***pd*** The protection domain to delete

Returns:

If called on your own protection domain this call will not return

E.3.1.8 pd_ref_t pd_myself (void)

Return a reference to the current protection domain.

E.3.1.9 void pd_set_callback (pd_ref_t *pd*, memsection_ref_t *callback*)

Setup an async communications channel for messages.

Parameters:

← ***pd*** The containing protection domain

← ***callback*** A reference to the callback buffer.

E.4 libs/iguana/include/iguana/eas.h File Reference

```
#include <l4/types.h>
#include <iguana/types.h>
```

Functions

- **eas_ref_t eas_create** (L4_Fpage_t kip, L4_Fpage_t utcb)

  Create an external address space.

- void **eas_delete** (**eas_ref_t** eas)

  Delete an external address space.

- L4_ThreadId_t **eas_create_thread** (**eas_ref_t** eas, L4_ThreadId_t pager, L4_ThreadId_t scheduler, void ∗utcb)

  Create a new thread in the external address space.

- void **eas_delete_thread** (**eas_ref_t** eas, L4_ThreadId_t thread)

E.4.1 Function Documentation

E.4.1.1 eas_ref_t eas_create (L4_Fpage_t *kip*, L4_Fpage_t *utcb*)

Create an external address space.

E.4.1.2 L4_ThreadId_t eas_create_thread (eas_ref_t *eas*, L4_ThreadId_t *pager*, L4_ThreadId_t *scheduler*, void ∗ *utcb*)

Create a new thread in the external address space.

Parameters:

← ***eas*** The external address space

← ***pager*** The new thread's pager

← ***scheduler*** The new thread's scheduler

E.4.1.3 void eas_delete (eas_ref_t *eas*)

Delete an external address space.

Parameters:
← ***eas*** the EAS to delete.

E.4.1.4 void eas_delete_thread (eas_ref_t *eas*, L4_ThreadId_t *thread*)

E.5 libs/iguana/include/iguana/object.h File Reference

```
#include <l4/types.h>
#include <iguana/types.h>
```

Classes

- struct **object_t**

Functions

- **object_t** ∗ **object_get_interface** (**objref_t** obj)
- **object_t** ∗ **object_get_async_interface** (**objref_t** obj)
- void **object_print** (**object_t** ∗instance)

E.5.1 Function Documentation

E.5.1.1 object_t∗ object_get_async_interface (objref_t *obj*)

E.5.1.2 object_t∗ object_get_interface (objref_t *obj*)

E.5.1.3 void object_print (object_t ∗ *instance*)

E.6 libs/iguana/include/iguana/session.h File Reference

```
#include <stdbool.h>
#include <l4/types.h>
#include <iguana/types.h>
```

Classes

- struct **session**

Functions

- **session** ∗ **session_create** (**objref_t** object, **memsection_ref_t** clist, L4_ThreadId_t ∗server_tid)
  Create a new session, with a specified clist.

- **session** ∗ **_session_create** (**objref_t** object, **memsection_ref_t** clist, L4_ThreadId_t ∗server_tid, struct **session** ∗**session**)
- **session** ∗ **session_create_new_clist** (**objref_t** object, L4_ThreadId_t ∗server_tid)

  Create a new session, and create a dedicated clist for it.
- **session** ∗ **session_create_full_share** (**objref_t** object, L4_ThreadId_t ∗server_tid)

  Create a new session, and share your base clist with it.
- void **session_add_async** (struct **session** ∗**session**, **objref_t** call_buf, **objref_t** return_buf)

  Set up async buffers for session communcations.
- bool **session_provide_access** (struct **session** ∗**session**, **objref_t** object, int interface)
- void **session_delete** (struct **session** ∗**session**)

E.6.1 Function Documentation

E.6.1.1 struct session∗ _session_create (objref_t *object*, memsection_ref_t *clist*, L4_ThreadId_t ∗ *server_tid*, struct session ∗ *session*)

E.6.1.2 void session_add_async (struct session ∗ *session*, objref_t *call_buf*, objref_t *return_buf*)

Set up async buffers for session communcations.

Parameters:

← ***session*** The session to use

← ***call_buf*** A circular buffer for calls through the session

← ***return_buf*** A circular buffer for return values from the session

E.6.1.3 struct session∗ session_create (objref_t *object*, memsection_ref_t *clist*, L4_ThreadId_t ∗ *server_tid*)

Create a new session, with a specified clist.

Parameters:

object The object that you wish to access

clist The clist which the server will be provided with.

server_tid The L4 thread Id of the approriate server is returned.

Returns:

A pointer to the userlevel session object

E.6.1.4 struct session∗ session_create_full_share (objref_t *object*, L4_ThreadId_t ∗ *server_tid*)

Create a new session, and share your base clist with it.

This is obviously not meant to be generally used, and provides a short cut until all software correctly uses caps.

Parameters:

$\leftarrow$ ***object*** The object that you wish to access

$\rightarrow$ ***server_tid*** The L4 thread Id of the approriate server is returned

Returns:

A pointer to the userlevel session object

E.6.1.5 struct session$*$ session_create_new_clist (objref_t *object*, L4_ThreadId_t $*$ *server_tid*)

Create a new session, and create a dedicated clist for it.

Parameters:

object The object that you wish to access

server_tid The L4 thread Id of the approriate server is returned

Returns:

A pointer to the userlevel session object

E.6.1.6 void session_delete (struct session $*$ *session*)

Delete a session making any communication based on the session impossible.

Parameters:

session The session to delete

E.6.1.7 bool session_provide_access (struct session $*$ *session*, objref_t *object*, int *interface*)

Provide a given session access to a specific interface on an object

Parameters:

session The session to provide the access to

object The object to provide access to

interface The allowed interface

Returns:

True if access was provided. False otherwise. You may not be able to provide access if you do not have access, or have access, but can not transfer it.

E.7 libs/iguana/include/iguana/thread.h File Reference

```
#include <l4/types.h>
#include <iguana/types.h>
```

Functions

- L4_ThreadId_t **thread_l4tid** (**thread_ref_t** server)

  Return the L4 global thread ID for this Iguana thread.

- **thread_ref_t thread_myself** (void)

  Retrieve the Iguana thread reference of the current thread.

- **thread_ref_t thread_create** (L4_ThreadId_t ∗thrd)

  Create a new thread in the current PD.

- **thread_ref_t thread_create_priority** (int priority, L4_ThreadId_t ∗thrd)

  Create a new thread in the current PD and assign it a non-default priority.

- void **thread_start** (**thread_ref_t** thread, uintptr_t ip, uintptr_t sp)

  Activate an inactive thread.

- void **thread_delete** (L4_ThreadId_t thrd)

  Delete a thread.

- **thread_ref_t thread_id** (L4_ThreadId_t thrd)

  Return the Iguana thread reference for a given L4 global thread Id.

E.7.1 Function Documentation

E.7.1.1 thread_ref_t thread_create (L4_ThreadId_t ∗ *thrd*)

Create a new thread in the current PD.

A new Iguana thread is created in the current protection domain. The thread is initially created inactive; use L4's ExchangeRegisters() function, or **thread_start()**(p. 45), below, to activate it.

Parameters:

$\rightarrow$ ***thrd*** The L4 global thread Id of the new thread is stored here

Returns:

the Iguana thread reference for the new thread

E.7.1.2 thread_ref_t thread_create_priority (int *priority*, L4_ThreadId_t ∗ *thrd*)

Create a new thread in the current PD and assign it a non-default priority.

Parameters:

$\leftarrow$ ***priority*** An integer between 1 (lowest) and 255 (highest) specifying the priority

$\rightarrow$ ∗***thrd*** The L4 global thread Id of the new thread is stored here

Returns:

the Iguana thread reference for the new thread

E.7.1.3 void thread_delete (L4_ThreadId_t *thrd*)

Delete a thread.

Parameters:

← ***thrd*** the L4 thread Id of the thread to delete

E.7.1.4 thread_ref_t thread_id (L4_ThreadId_t *thrd*)

Return the Iguana thread reference for a given L4 global thread Id.

Parameters:

← ***thrd*** the L4 thread Id

Returns:

the Iguana thread reference for "thrd"

E.7.1.5 L4_ThreadId_t thread_l4tid (thread_ref_t *server*)

Return the L4 global thread ID for this Iguana thread.

Parameters:

← ***server*** The Iguana thread for which an L4 global ID is desired

Returns:

The L4 global thread ID for this thread

E.7.1.6 thread_ref_t thread_myself (void)

Retrieve the Iguana thread reference of the current thread.

Returns:

The Iguana thread reference of the current thread

E.7.1.7 void thread_start (thread_ref_t *thread*, uintptr_t *ip*, uintptr_t *sp*)

Activate an inactive thread.

Make an inactive thread schedulable. The thread will start executing at the supplied instruction pointer.

Parameters:

← ***thread*** the Iguana thread reference to the thread to activate

← ***ip*** the initial IP address of the thread

← ***sp*** the initial stack pointer of the thread

Appendix F

C Bindings for Low-level API

Appendix G

Iguana IDL

Iguana uses a subset of CORBA IDL format to define interfaces. The IDL file is processed to produce two *stub files*, one for the server-side and one for the client-side. For reference, Iguana's main IDL file is reproduced below.

```
 *
 * EXCEPT AS EXPRESSLY STATED IN THIS LICENCE AND TO THE FULL EXTENT
 * PERMITTED BY APPLICABLE LAW, THE SOFTWARE IS PROVIDED "AS–IS", AND
 * NATIONAL ICT AUSTRALIA AND ITS CONTRIBUTORS MAKE NO REPRESENTATIONS,
 * WARRANTIES OR CONDITIONS OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
 * BUT NOT LIMITED TO ANY REPRESENTATIONS, WARRANTIES OR CONDITIONS
 * REGARDING THE CONTENTS OR ACCURACY OF THE SOFTWARE, OR OF TITLE,
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT,
 * THE ABSENCE OF LATENT OR OTHER DEFECTS, OR THE PRESENCE OR ABSENCE OF
 * ERRORS, WHETHER OR NOT DISCOVERABLE.
 *
 * TO THE FULL EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL
 * NATIONAL ICT AUSTRALIA OR ITS CONTRIBUTORS BE LIABLE ON ANY LEGAL
 * THEORY (INCLUDING, WITHOUT LIMITATION, IN AN ACTION OF CONTRACT,
 * NEGLIGENCE OR OTHERWISE) FOR ANY CLAIM, LOSS, DAMAGES OR OTHER
 * LIABILITY, INCLUDING (WITHOUT LIMITATION) LOSS OF PRODUCTION OR
 * OPERATION TIME, LOSS, DAMAGE OR CORRUPTION OF DATA OR RECORDS; OR LOSS
 * OF ANTICIPATED SAVINGS, OPPORTUNITY, REVENUE, PROFIT OR GOODWILL, OR
 * OTHER ECONOMIC LOSS; OR ANY SPECIAL, INCIDENTAL, INDIRECT,
 * CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES, ARISING OUT OF OR IN
 * CONNECTION WITH THIS LICENCE, THE SOFTWARE OR THE USE OF OR OTHER
 * DEALINGS WITH THE SOFTWARE, EVEN IF NATIONAL ICT AUSTRALIA OR ITS
 * CONTRIBUTORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH CLAIM, LOSS,
 * DAMAGES OR OTHER LIABILITY.
 *
 * If applicable legislation implies representations, warranties, or
 * conditions, or imposes obligations or liability on National ICT
 * Australia or one of its contributors in respect of the Software that
 * cannot be wholly or partly excluded, restricted or modified, the
 * liability of National ICT Australia or the contributor is limited, to
 * the full extent permitted by the applicable legislation, at its
 * option, to:
 * a.  in the case of goods, any one or more of the following:
 * i.  the replacement of the goods or the supply of equivalent goods;
 * ii.  the repair of the goods;
 * iii. the payment of the cost of replacing the goods or of acquiring
 *  equivalent goods;
 * iv.  the payment of the cost of having the goods repaired; or
 * b.  in the case of services:
 * i.  the supplying of the services again; or
 * ii.  the payment of the cost of having the services supplied again.
 *
 * The construction, validity and performance of this licence is governed
 * by the laws in force in New South Wales, Australia.
 */
/*
 * Iguana IDL for dealing with userland
 * eg. pagefaults, exceptions, syscalls.
 *
 */
/* Import L4 and standard types */
```

```
import "l4/types.h";
import "iguana/types.h";
import "stdint.h";
import "stddef.h";
/* FIXME: define these numbers some place sane for manageability */
[uuid(21)]
interface iguana_ex
{
        /* a pagefault (from iguana userland) */
        [kernelmsg(idl4::pagefault)]
        void pagefault(in uintptr_t addr, in uintptr_t ip, in uintptr_t priv,
                       out fpage fp);

        /* FIXME: add thread exceptions */
};
[uuid(22)]
interface iguana_pd
{
        /* This method is kind of magic */
        objref_t mypd();
        cap_t create_memsection(in pd_ref_t pd, in uintptr_t size, in uintptr_t base, in int flags,
                                out uintptr_t base_out);
        cap_t create_pd(in pd_ref_t pd);
        cap_t create_thread(in pd_ref_t pd, in int priority, out L4_ThreadId_t l4_id);
        cap_t create_eas(in pd_ref_t pd, in L4_Fpage_t kip, in L4_Fpage_t utcb);
        cap_t create_session(in pd_ref_t pd, in thread_ref_t client, in thread_ref_t server,
                             in memsection_ref_t clist);
        void set_callback(in pd_ref_t pd, in memsection_ref_t callback_buffer);
        uintptr_t add_clist(in pd_ref_t pd, in memsection_ref_t clist);
        void delete(in pd_ref_t pd);
};
[uuid(23)]
interface iguana_eas
{
        cap_t create_thread(in eas_ref_t eas, in L4_ThreadId_t pager,
                            in L4_ThreadId_t scheduler, in uintptr_t utcb,
                            out L4_ThreadId_t l4_id);
        void delete(in eas_ref_t eas);
};
[uuid(24)]
interface iguana_thread
{
        /* This is another ‘magic’ method */
        thread_ref_t id(in L4_ThreadId_t thread);
        L4_ThreadId_t l4id(in thread_ref_t thread);
        void start(in thread_ref_t thread, in uintptr_t ip, in uintptr_t sp);
/*      pd_ref_t domain(in thread_ref_t thread);*/
        void delete(in thread_ref_t thread);
};
[uuid(25)]
interface iguana_hardware
```

```
{
        int register_interrupt(in hw_ref_t hardware, in L4_ThreadId_t handler,
                                         in int interrupt);
        int back_memsection(in hw_ref_t hardware, in memsection_ref_t memsection,
                                      in uintptr_t paddr, in uintptr_t attributes);
};
[uuid(26)]
interface iguana_memsection
{
        int register_server(in memsection_ref_t memsection, in thread_ref_t thread);
        memsection_ref_t lookup(in uintptr_t address, out thread_ref_t server);
        uintptr_t info(in memsection_ref_t memsection);
        void delete(in memsection_ref_t thread);
};
[uuid(27)]
interface iguana_session
{
        void delete(in session_ref_t session);
        void add_buffer(in session_ref_t session, in objref_t call_buf, in objref_t return_buf);
};
```

Bibliography