ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik Computer Science

Advanced Operating Systems (263-3800-00L) Programming L4

Timothy Roscoe & Andrew Baumann

Based on slides by Kevin Elphinstone & Gernot Heiser (UNSW).

L4 Introduction

- "Second generation" µkernel
- Ports to many architectures, active development
  - Karlsruhe, Dresden, Sydney, ...
  - In use commercially (in mobile phones)
- Also the basis of your lab project!
  - You'll be using the NICTA N2 version

L4 Introduction

- Minimum features: address spaces, threads, IPC

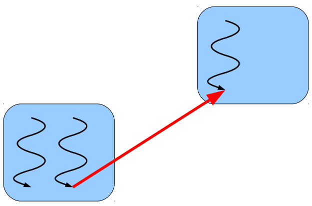
  "A feature is only allowed in the kernel if this is required for the implementation of a secure system."
- Very fast IPC
  - Orders of magnitude faster than older µkernels
  - Due to small cache footprint
- Small kernel (~10kLOC) has other advantages:
  - Small trusted computing base
  - Verifiable kernel implementation

Overview

- L4 introduction: abstractions and mechanisms
- Threads and thread management
  - **`ThreadControl`**, **`ExchangeRegisters`**
- IPC
  - **`IPC`**, Interrupt protocol
- Scheduling
  - **`ThreadSwitch`**, **`Schedule`**
- Address space management
  - **`MapControl`**, Page fault protocol
- Misc
  - **`SpaceControl`**, **`CacheControl`**
  - Preemption and exception protocols

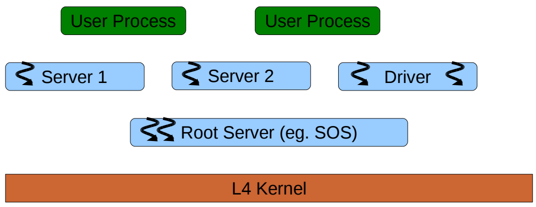
L4 Abstractions and Mechanisms

- Address spaces
  - Unit of protection
- Threads
  - Unique identifiers
  - Execution abstraction
- Mapping
  - For address-space management



- IPC
  - (mostly) synchronous

Example L4-based system



- Multi-server decomposition of monolithic kernel
  - Each server is in a separate HW address space
  - For the project, consider something simpler

L4 Abstractions: Address Spaces

- Address space is unit of protection in L4
  - Initially empty
  - Populated by mapping in frames
- Constructed by privileged *MapControl* syscall
  - Map/unmap operations
  - Can only be called by *root task*

L4 Concepts: Root Task

- First address space created at boot time
- Can perform privileged system calls
- Controls system resources
  - Threads
  - Address spaces
  - Physical memory
- Cannot delegate privilege
  - Shortcoming of X.2 and N1/N2 APIs

L4 Abstractions: Threads

- Kernel-scheduled unit of execution
- Every thread has a unique ID
  - Identifies threads for IPC and other kernel operations
- Threads managed by user-level servers
  - Creation, destruction, association with address space
- Other thread attributes:
  - Scheduling parameters (time slice, priority)
  - Page-fault and exception handlers

L4 Abstractions: Time

- Used for scheduling time slices
  - Thread has fixed-length time slice for preemption
  - Time slices allocated from (finite or infinite) time quantum
    - Notification when exceeded
- Also used for IPC timeouts in other L4 APIs

L4 Mechanism: IPC

- Synchronous message-passing operation
  - N2 API also has asynchronous variant
- Data copied directly from sender to receiver
  - Short messages passed in registers to avoid copying
- Can be blocking or polling
  - Blocking: don't return until transfer occurs
  - Polling: fails immediately if partner not ready
- Exceptions modelled as IPC

L4 System Calls (in N2 API)

- `KernelInterface`
- `ThreadControl`
- `ExchangeRegisters`
- `IPC`
- `ThreadSwitch`
- `Schedule`
- `MapControl`
- `SpaceControl`
- `ProcessorControl`
- `CacheControl`

Red = privileged syscall

Kernel Information Page (KIP)

- Kernel-defined memory object located in every address space
  - Placed on address space creation
    - Location dictated by **`SpaceControl`** system call
- Contains information about kernel and hardware
  - Supported page sizes
  - API version
  - Physical memory layout
  - Addresses of system call functions

~	SCHEDULE *SC*	THREADSWITCH *SC*	*Reserved*	+F0 / +1E0
EXCHANGEREGISTERS *SC*	LIPC *SC*	IPC *SC*	*Reserved*	+E0 / +1C0
PROCESSORCONTROL *pSC*	THREADCONTROL *pSC*	SPACECONTROL *pSC*	MAPCONTROL *pSC*	+D0 / +1A0
ProcessorInfo	PageInfo	ThreadInfo	ClockInfo	+C0 / +180
ProcDescPtr	BootInfo	~		+B0 / +160
KipAreaInfo	UtcbInfo	VirtualRegInfo	~	+A0 / +140
~				+90 / +120
~				+80 / +100
~				+70 / +E0
~				+60 / +C0
~		MemoryInfo	~	+50 / +A0
~				+40 / +80
~				+30 / +60
~				+20 / +40
~				+10 / +20
KernDescPtr	API Flags	API Version	$0_{(0/32)}$ 'K' 230 '4' 'L'	+0
+C / +18	+8 / +10	+4 / +8	+0	

`KernelInterface()`

- "Magic" system call to locate the KIP
- Defined to be slow, so result should be cached
  - libl4 does this for you
- C API prototype:

```
void *L4_KernelInterface(L4_Word_t *ApiVersion
                         L4_Word_t *ApiFlags,
                         L4_Word_t *KernelId);
```

- Most calls are hidden by libl4 for convenience

A note on source code

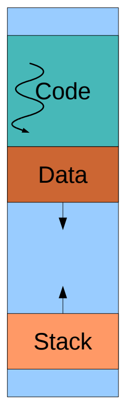
- This lecture covers the lowest-level syscall API
- Many calls also have convenience wrappers in the libl4 library
  - Documented in API reference manual
- In addition to the kernel and libl4, we provide some additional code and utility functions
  - sos/libsos.[ch] in the project source
  - Best documentation is the code itself

Overview

- **Threads and thread management**
  - **`ThreadControl`**, **`ExchangeRegisters`**
- IPC
  - **`IPC`**, Interrupt protocol
- Scheduling
  - **`ThreadSwitch`**, **`Schedule`**
- Address space management
  - **`MapControl`**, Page fault protocol
- Misc
  - **`SpaceControl`**, **`CacheControl`**
  - Preemption and exception protocols

Traditional Thread

- Abstraction and unit of execution
- Consists of:
  - Registers, including:
    - Instruction pointer
    - Stack pointer
    - Processor status
  - Stack
    - Execution history of unreturned procedures
    - One *stack frame* per procedure call



L4 Thread

- An L4 thread is a traditional thread plus:
  - A set of *virtual registers*
  - Scheduling parameters (priority, timeslice)
  - Unique thread identifier
  - An address space
    - May be shared with other threads
- L4 provides a fixed overall number of threads
  - Root task responsible for creating/deleting threads
  - System, user and “interrupt” threads

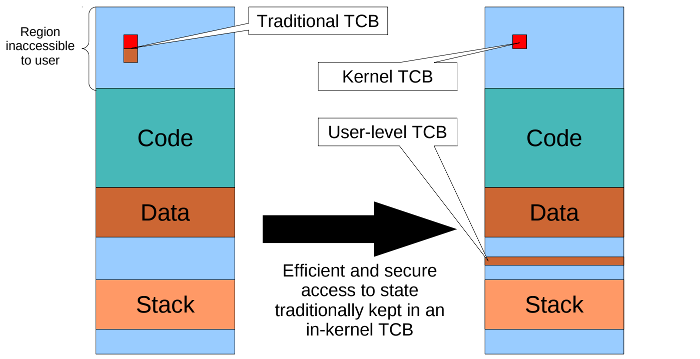
Virtual Registers

- Per-thread “registers” defined by L4
- May be real CPU registers or memory locations
  - Depends on architecture and ABI
- Two basic types (in N2):
  - Thread control registers (TCRs)
    - Used to share thread information between kernel and user-level
  - Message registers (MRs)
    - Used to send messages between threads; contains message data
    - Also used for arguments to `MapControl`

Thread Control Blocks (TCBs)

- State of a thread is stored in its TCB
- Sensitive state may only be modified via syscalls
  - Eg: address space, stack pointer
  - Stored in Kernel TCB (KTCB)
- Other state is accessible (R/W) to user-level code without compromising the system
  - Stored in User-level TCB (UTCB)
  - Includes virtual registers not bound to a CPU register

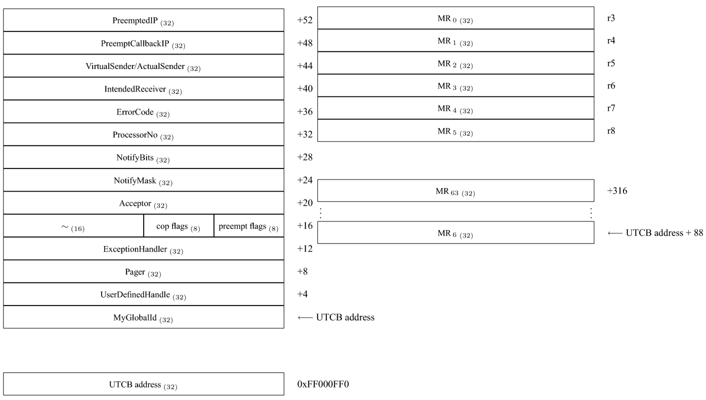
KTCBs and UTCBs



UTCB Programming

- Must only be modified via provided programming interface (libl4)
  - No consistency guarantees otherwise, so don't access it directly
  - Library function scope is limited to the current thread
- You can mostly ignore its contents
  - Most fields are set/read in the context of other actions (eg. IPC) or modified by the kernel as a side-effect

ARM UTCB Layout



ThreadControl()

- Used to create, destroy, or modify threads
- Determines thread attributes:
  - Thread identifier
  - Address space
  - Another thread allowed to set scheduling parameters
    - Note: the "scheduler" thread does not perform CPU scheduling
  - The thread's page-fault handler ("pager")
  - Location of the thread's UTCB
    - Not on ARM however: the UTCB address is defined by the kernel

ThreadControl()

- C API:

```
L4_Word_t L4_ThreadControl(
                L4_ThreadId_t dest,
                L4_ThreadId_t SpaceSpecifier,
                L4_ThreadId_t Scheduler,
                L4_ThreadId_t Pager,
                L4_ThreadId_t SendRedirector,
                L4_ThreadId_t RecvRedirector,
                void *UtcbLocation);
```

  - Note: on ARM, **UtcbLocation** must be NULL when activating a thread

ThreadControl()

- Threads may be created *active* or *inactive*
  - Thread is created active iff it has a pager set
  - Creation of inactive threads is used to:
    - Create and manipulate new address spaces
    - Allocate new threads to existing address spaces
  - Inactive threads may later be activated by one of:
    - A privileged thread, using **ThreadControl**
    - Any thread in the same address space, using **ExchangeRegisters**

Thread Identifiers

- Uniquely identify a thread
- Defined by root task at thread creation time
  - According to some policy
  - Constraints:
    - $\text{Version}_{[5..0]} \neq 0$
    - Thread No ≥ **UserBase** (see KIP)
    - Thread No ≠ -1

Global Thread ID:

$\text{Thread No}_{(18)}$	$\text{Version}_{(14)}$

Global Interrupt ID:

$\text{Interrupt No}_{(18)}$	$1_{(14)}$

Task

- The word "task" is often used informally to mean:
  - An address space
    - UTCB area
    - KIP location
    - Redirector
  - The set of threads inside it; each has:
    - Thread ID
    - UTCB location
    - IP, SP, other state
    - Pager
    - Scheduler
    - Exception handler
  - Code, data and stacks mapped into the address space

Steps in creating a new task

1. Create inactive thread in a new address space:
   - An address space is referred to via one of its threads

```
r = L4_ThreadControl(task, /* new thread ID */
                     task, /* new space ID */
                     me,   /* scheduler of new thread */
                     L4_nilthread, /* no pager: inactive */
                     L4_anythread, /* no send redirector */
                     L4_anythread, /* no receive redirector */
                     (void *) -1); /* no UTCB location */
```

Steps in creating a new task

2. Define the locations of the KIP and UTCB area in the new address space:

```
r = L4_SpaceControl(task,      /* new thread ID */
                    0,         /* control */
                    kip_area,  /* where KIP is mapped */
                    utcb_area, /* location of UTCB array */
                    &control); /* leave alone */
```


Steps in creating a new task

3. Specify the UTCB location and assign a pager to the new thread to activate it:

```
r = L4_ThreadControl(task, task, me,
                     pager,      /* new pager */
                     utcb_base); /* UTCB location */
```

- This activates the new thread and sets it waiting for an IPC message containing its initial IP and SP
- Note: on ARM, **`utcb_base`** must be NULL

Steps in creating a new task

4. Send an IPC to the new thread with its initial IP and SP in the first two words of the message.
   - This results in the new thread starting execution at the received IP with the SP set as in the message.

Adding threads to a task

- Use **`ThreadControl`** to assign a new thread:

```
r = L4_ThreadControl(newtid,      /* new thread ID */
                     existingtid, /* address space ID*/
                     me,          /* scheduler */
                     pager,       /* pager */
                     L4_anythread, L4_anythread,
                     utcb_base);  /* as before */
```

- Can also create the new thread as inactive
  - Task can then manage new threads itself
  - ... using **`ExchangeRegisters`** syscall

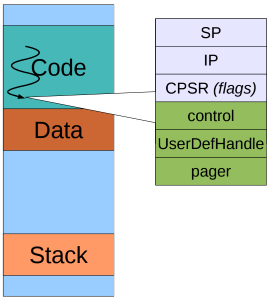
Practical Considerations

- Thread/task creation is cumbersome
  - Result of leaving policy out of the kernel
- A system built on top of L4 will define policies
  - Can implement library for task and thread creation
  - Incorporates system policy
  - See **`sos_task_new()`** for an example
- Actual apps would not use raw L4 system calls
  - Rather, libraries or IDL compilers

Manipulating threads within an AS

- So far we can:
  - Create a new address space with a single thread
  - Assign new threads to an existing address space
- **`ExchangeRegisters`** system call
  - Used to activate or modify an existing thread from within the same address space

ExchangeRegisters()




```
L4_ThreadId_t L4_ExchangeRegisters(L4_ThreadId_t dest,
                                   L4_Word_t control,
                                   L4_Word_t sp,
                                   L4_Word_t ip,
                                   L4_Word_t flags,
                                   L4_Word_t UserDefHandle,
                                   L4_ThreadId_t pager,
                                   L4_Word_t *old_control,
                                   L4_Word_t *old_sp,
                                   L4_Word_t *old_ip,
                                   L4_Word_t *old_flags,
                                   L4_Word_t *old_UserDefHandle,
                                   L4_ThreadId_t *old_pager);
```


ExchangeRegisters()

- Bits of ARM CPSR register affected by **`flags`**

N Z C V Q	$\sim_{(21)}$	T	$\sim_{(5)}$

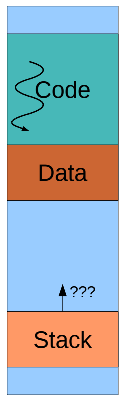
N: negative
Z: zero
C: carry
V: overflow
Q: sticky overflow
T: thumb mode

Thread management

- L4 usually only preserves the user's IP and SP
  - ... and full register state if thread is preempted
- The following are managed by user-level (you!):
  - Stack (location, allocation, size)
  - Thread ID (allocation, deallocation)
  - Entry point IP

Stack corruption

- This is a common problem
- Stack corruption is very difficult to:
  - Diagnose
  - Debug
- Be careful!



Overview

- ✓ Threads and thread management
  - **`ThreadControl`**, **`ExchangeRegisters`**
- → **IPC**
  - **`IPC`**, Interrupt protocol
- Scheduling
  - **`ThreadSwitch`**, **`Schedule`**
- Address space management
  - **`MapControl`**, Page fault protocol
- Misc
  - **`SpaceControl`**, **`CacheControl`**
  - Preemption and exception protocols

IPC Overview

- L4 provides a single system call for all IPC
  - Synchronous and unbuffered *(apart from async notify)*
  - Has a send and a receive component
  - Either send or receive may be omitted
- Receive may specify:
  - A specific thread ID from which to receive (“closed receive”)
  - Willingness to receive from any thread (“open wait”)

Logical IPC operations

- **`Send`** sends a message to a specific thread
- **`Receive`** “closed” receive from a specific sender
- **`Wait`** “open” receive from any sender
- **`Call`** send to and wait for reply from specific thread
  - Typical client RPC operation
- **`ReplyWait`** send to specific thread, “open” receive
  - Typical server operation

Special Thread Identifiers

- A thread ID may be one of:
  - User thread ID
  - Interrupt ID
  - Nil thread
  - Any thread
  - Wait notify

User thread ID	Thread No$_{(18)}$	Version$_{(14)}$
Interrupt ID	Interrupt No$_{(18)}$	1$_{(14)}$
Nil thread	0$_{(32)}$	
Any thread	-1$_{(32)}$	
Wait notify	-2$_{(32)}$	

```
#define L4_nilthread        ((L4_ThreadId_t) { raw : 0UL})
#define L4_anythread        ((L4_ThreadId_t) { raw : ~0UL})
#define L4_waitnotify       ((L4_ThreadId_t) { raw : -2UL})
```


IPC Message Registers (MRs)

- Virtual registers
  - Not necessarily backed by CPU registers
  - Part of thread state
  - On ARM: 6 physical registers, rest in UTCB
- Actual number is a system configuration parameter
  - At least 8, no more than 64
- Contents of MRs form message
  - First MR stores the *message tag* defining message size etc.
  - Rest are untyped words, not normally interpreted by the kernel
  - Kernel protocols define semantics in some cases
- IPC just copies data from sender's to receiver's MRs

IPC Message Transfer

Sender		Receiver
MR_{63}		MR_{63}
⋮		⋮
MR_7	Message transferred (copied) from sender thread's MRs to receiver thread's MRs. →	MR_7
MR_6		MR_6
MR_5		MR_5
MR_4		MR_4
MR_3		MR_3
MR_2		MR_2
MR_1		MR_1
MR_0		MR_0

- Guaranteed not to page fault (registers don't fault!)
- Highly optimised in L4 kernel (“fast path”)

Message tag: MR_0

label$_{(16)}$	s	r	n	p	~$_{(6)}$	u$_{(6)}$

- u: number of words in message (excluding tag)
- p: specifies *propagation*
  - Allows sending on behalf of another thread; details in L4 manual
- n: specifies *asynchronous notification* operation
- r: blocking receive
  - If unset, fail immediately if no message is pending
- s: blocking send
  - If unset, fail immediately if receiver is not waiting
- label: user-defined value (eg. opcode, syscall number)

IPC Example: sending 4 words

- Only 5 MRs transferred
  - Note: on ARM, 6 MRs are transferred in registers
    - Fast/optimised
    - The rest, if used, are copied from/to UTCB memory

Note:

- u, s, r set implicitly by **`L4_MsgAppendWord()`**
- Ideally we would use an IDL compiler instead of manually generating messages

MR_5 (unused)						
Word 4						
Word 3						
Word 2						
Word 1						
label$_{(16)}$	0	0	0	0	0$_{(6)}$	4$_{(6)}$

IPC Example: sending 4 words

```
L4_Msg_t msg;
L4_MsgTag_t tag;

L4_MsgClear(&msg);
L4_MsgAppendWord(&msg, word1);
L4_MsgAppendWord(&msg, word2);
L4_MsgAppendWord(&msg, word3);
L4_MsgAppendWord(&msg, word4);
L4_Set_MsgLabel(&msg, label);
L4_MsgLoad(&msg);

tag = L4_Send(dest_tid);
```

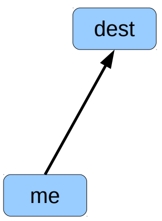

IPC Result Tag: MR_0

label$_{(16)}$	E	X	r	p	~$_{(6)}$	u$_{(6)}$

- u: number of words received (u = 0 for send-only IPC)
- p: received propagated IPC
  - Check **ActualSender** field in UTCB
- r: received redirected IPC
  - Check **IntendedReceiver** field in UTCB
- X: received cross-processor IPC
  - Shouldn't happen!
- E: error indicator
  - If non-zero, check **ErrorCode** field in UTCB for details

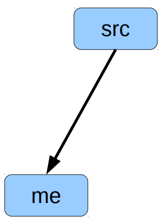
IPC Send

to:	dest
FromSpecifier:	*nil*



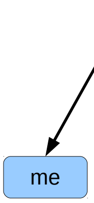
IPC Receive (closed)

to:	*nil*
FromSpecifier:	src



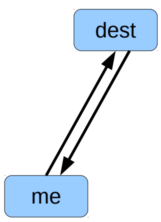
IPC Wait (open)

to:	*nil*
FromSpecifier:	*any*



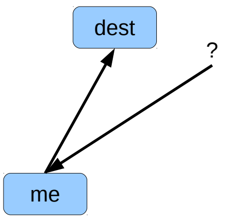
IPC Call

to:	dest
FromSpecifier:	dest



IPC ReplyWait

to:	dest
FromSpecifier:	*any*



Asynchronous Notification

Very restricted form of asynchronous IPC

- Delivered without blocking sender
- Delivered immediately, directly to receiver's UTCB
- Message consists of a bit mask ORed to the receiver:
  receiver.**`NotifyBits`** |= sender.MR_1
- No effect if receiver's bits are already set
- Receiver can prevent asynchronous notification by setting a flag in its UTCB

Receiving Asynchronous Notifications

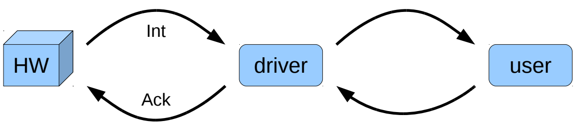
- *Synchronously*, by a form of blocking IPC wait
  - Receiver specifies mask of notification bits to wait for
  - On notification, kernel manufactures reply message
- *Asynchronously*, using **`NotifyBits`** in UTCB
  - ... but remember it's asynchronous and can change at any time

L4 Exception Handling: Interrupts

- Modelled as hardware “thread” sending messages
- Received by registered (user-level) handler thread
  - Usually a device-driver which also has a mapping for the device memory
- Acknowledged (reenabled) when handler thread blocks on receive

Device Interrupts

- Modelled as messages from “hardware” threads
- Acknowledged by empty reply message (MR_0=0)



- IO ports mapped to the driver's address space

Interrupt Association

- Association performed by privileged thread (root task) calling **`ThreadControl`**
- To associate a thread to an interrupt:
  - Set the interrupt thread's pager to the handler thread
- To dissociate a thread from an interrupt:
  - Set the interrupt thread's pager to itself

Interrupt Handlers

- Typical setup: handler is bottom-half driver

1. Interrupt triggered, CPU disables interrupts and invokes kernel
2. Kernel masks interrupt, re-enables interrupts, sends message to handler
3. Handler receives message, identifies cause, and replies
4. Kernel acknowledges (unmasks) interrupt
5. Handler queues request to top-half driver, sends notification to top half, and waits for next message

Overview

- Threads and thread management
  - `ThreadControl`, `ExchangeRegisters`
- IPC
  - `IPC`, Interrupt protocol
- **Scheduling**
  - `ThreadSwitch`, `Schedule`
- Address space management
  - `MapControl`, Page fault protocol
- Misc
  - `SpaceControl`, `CacheControl`
  - Preemption and exception protocols

`ThreadSwitch()`

Forfeits caller's remaining time slice

- May donate remaining time slice to specific thread
  - Target will execute to the end of time slice on donor's priority
  - *Or actually until the next timer tick, in the current implementation*
- If no recipient specified (or recipient is not runnable)
  - Normal "yield" operation
  - Kernel invokes scheduler
  - Caller might immediately receive a new time slice
- Directed donation may be used for explicit scheduling, wait-free locks, ...

L4 Scheduling

- L4 provides 256 hard priority levels (0-255)
- Within each priority, threads are scheduled round-robin
- Scheduler is invoked when:
  - The current thread is pre-empted
  - The current thread yields
- Scheduler is **not** normally invoked when a thread blocks
  - If destination of an IPC is runnable, the kernel will switch to it
  - Scheduler invoked only if destination is blocked too
  - If both threads are runnable after IPC, the higher priority one *should* run
- Designed to avoid (expensive) scheduler invocations

`Schedule()`

Manipulates a thread's scheduling parameters

- NB: does not invoke a scheduler nor schedule threads
- Caller must be registered as the destination's scheduler (via **`ThreadControl`**)
- Can change:
  - Priority
  - Timeslice duration
  - Total quantum
  - Processor number (for thread migration on an MP)

Overview

- Threads and thread management
  - `ThreadControl`, `ExchangeRegisters`
- IPC
  - `IPC`, Interrupt protocol
- Scheduling
  - `ThreadSwitch`, `Schedule`
- **Address space management**
  - `MapControl`, Page fault protocol
- Misc
  - `SpaceControl`, `CacheControl`
  - Preemption and exception protocols

Address Spaces

- Created empty
- Need to be explicitly populated with mappings
  - L4 won't map pages (except KIP/UTCB) automatically
- Normally populated on demand by a pager
  - Thread runs, faults on unmapped page, pager creates new mapping
- May be pre-populated
  - Eg. OS server can pre-map contents of an executable

`MapControl()`

Privileged call, creates or destroys page mappings.

```
L4_Word_t L4_MapControl(L4_ThreadId_t dest,
                        L4_Word_t control);
```

- **`dest`**: identifies target address space
- **`control`**:

m	r	$0_{(24)}$	$n_{(6)}$

  - r: read operation, returns (pre-syscall) mapping info
    - Eg. reference bits where hardware maintained (x86)
  - m: modify operation, changes mappings
  - n: number of *map items* used to describe mappings
    - Stored in message registers MR_0-MR_{2n-1}

FPages

A *flexpage* specifies an address space region

- Generalisation of a hardware page, similar properties:
  - Size is power-of-two multiple of hardware base page size
  - Must be naturally aligned
- fpage of size 2^s is specified as:

$base/1024_{(22)}$	$s_{(6)}$	$\sim_{(4)}$

- Special fpages:
  - Full AS:

$0_{(22)}$	$0x3f_{(6)}$	$\sim_{(4)}$

  - Nil fpage:

$0_{(22)}$	$0_{(6)}$	$0_{(4)}$

- On ARM, $s \geq 12$ (2^{12} = 4KiB pages)

Map Items

Specifies a mapping to be created in the target AS

$fpage_{(28)}$	0rwx
$phys/1024_{(26)}$	$attr_{(6)}$

- fpage: specifies where mapping is to appear
- phys: base of physical frame(s) to be mapped
  - Must be aligned to fpage size
  - Shifted 4 bits to support 64MiB of physical address space
- attr: memory attributes (eg. cached/uncached)
- rwx: access rights in destination address space
  - Can be used to change (up or downgrade) rights
  - Removing all rights removes the mapping (unmap operation)

L4 Exception Handling: Page Faults

- Kernel fakes IPC from faulting thread to its pager
- Pager (usually) requests root task to setup a mapping
  - Alternatively, if pager is in root task
- Pager replies to faulting thread, causing it to resume execution

Page Fault Handling

- Address spaces are (usually) populated in response to page fault IPC messages
  1. Application thread triggers page fault
  2. Kernel generates IPC from faulting thread to pager
  3. Pager establishes mapping if privileged, or contacts root task to do it (**`MapControl`**)
  4. Pager replies to fault IPC with null message
  5. Kernel intercepts message, discards, restarts thread

Page Fault Message

- Format of kernel-generated page-fault message:

Fault IP					MR_2
Fault address					MR_1
$-2_{(12)}$	0rwx	$0_{(4)}$	$\sim_{(6)}$	$2_{(6)}$	MR_0

- Application can manufacture the same message
  - Pager cannot tell the difference
  - Not a problem, as application could achieve the same by forcing a fault

Pager Action

- After establishing mapping, pager replies to fault
  - Content of message ignored
  - Serves for synchronisation: informs kernel that faulting thread can be restarted
  - If the pager did not establish a suitable mapping, the client will trigger the same fault again

Overview

- Threads and thread management
  - `ThreadControl`, `ExchangeRegisters`
- IPC
  - `IPC`, Interrupt protocol
- Scheduling
  - `ThreadSwitch`, `Schedule`
- Address space management
  - `MapControl`, Page fault protocol
- **Misc**
  - **`SpaceControl`**, **`CacheControl`**
  - Preemption and exception protocols

`SpaceControl()`

- Controls layout of new address spaces
  - KIP & UTCB area locations (not on ARM)
- Controls setting of *redirector*
  - Limits communication, for information flow control
  - If set to a valid thread ID, IPC may only be sent:
    - Locally (within the same address space)
    - To the redirector's address space
  - Other messages are instead delivered to the redirector

`ProcessorControl()`

- Sets processor core voltage and frequency
  - Where supported
  - Used for power management
- Privileged system call

`CacheControl()`

- Used to flush caches or lock cache lines as per arguments:
  - Target cache (I/D, L1/L2, ...)
  - Kind of operation (flush/lock/unlock)
  - Address range affected

L4 Protocols

- ✔ Thread start
- ✔ Interrupt
- ✔ Page fault
- ➔ Exception
- ➔ Preemption (not covered)

L4 Exception Handling: Other Exceptions

- Kernel fakes IPC from thread to its exception handler
- Exception handler may reply with message specifying new IP & SP
  - New IP could be a signal handler, emulation code, stub for IPCing to another server, etc.
- Interrupt, page fault and exception protocols documented in L4 reference manual

Exception Protocol

- Non-page-fault exceptions result in a kernel-generated IPC to the thread's *exception handler*
  - Invalid instruction, division by zero, etc.
- Exception IPC
  - Kernel sends partial thread state in MRs (MR_0 = IP)
  - Message label:
    - -4: standard (architecture independent) exceptions
    - -5: architecture-specific exception
  - Exception handler may reply with modified state

Exception Handling

Possible responses of exception handler:

- *Kill thread: use* ***`ExchangeRegisters`*** *or* ***`ThreadControl`***
- *Ignore*: blocks the thread indefinitely
- *Retry*: reply with unchanged state
  - Possibly after removing cause or changing other parts of state
- (…)

Preemption Protocol

- Each thread has three scheduling attributes:
  - Priority
  - Time slice duration
  - Total quantum
- When a thread is scheduled (according to priority):
  - It is given a fresh time slice
  - The time slice is deducted from its total quantum
- When a thread's total quantum is exhausted:
  - The kernel sends a message on its behalf to the thread's scheduler:

$-3_{(12)}$	$0_{(4)}$	$0_{(4)}$	$\sim_{(6)}$	$0_{(6)}$	MR_0

  - Scheduler may provide a new quantum (using **`Schedule`**)