L4 Programming Introduction

Fundamental Concepts

- Address Spaces
  - Unit of protection, resource management.
- Threads
  - Execution abstraction and provide unique identifiers.
- Time
  - Actual abstraction varies between API versions of L4.
- Communication: IPC
  - Synchronous
  - Identification: thread ids
  - AS construction: mapping
  - Flexpages
  - Architecture independent page abstraction

L4 System Calls

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

- 10 System Calls
  - N2 API – other API version vary
  - 6-8 Kernel defined protocols

3L4 System Calls

Root Task

- First task started at boot time
- Can perform privileged system calls
- Controls access to resources managed by privileged systems calls
  - ThreadControl, SpaceControl, ProcessorControl, MemoryControl
    - Thread allocation, memory attributes, processor modes, etc.

Kernel Information Page

- Kernel memory object located in the address space of a task
  - Placed on address space creation
    - Location is dictated by SpaceControl system call
- Contains information about kernel and hardware
  - Page sizes supported
  - API version
  - Physical memory layout,
  - Syscall Addresses

Kernel Information Page

- Kernel memory object located in the address space of a task
  - Placed on address space creation
    - Location is dictated by SpaceControl system call
- Contains information about kernel and hardware
  - Page sizes supported
  - API version
  - Physical memory layout,
  - Syscall Addresses
**KernelInterface()**

- System call provided to locate the kernel information page
- In ‘C’ api
  
  ```c
  void * L4_KernelInterface (L4_Word_t *ApiVersion,
  L4_Word_t *ApiFlags,
  L4_Word_t *KernelId)
  ```

---

**Traditional Thread**

- Abstraction and unit of execution
- Consists of:
  - Registers
    - Current variables
    - Status
  - Instruction Pointer
    - Next instruction to execute
  - Stack
    - Execution history of yet unretumed procedures
    - One stack frame per procedure invocation

---

**L4 thread = trad. thread +**

- A set of virtual register
- A priority and a timeslice
- A unique thread identifier
- An address space
- L4 provides a fixed overall number of threads in the system
  - Root task responsible for creating/deleting threads and assigning them to address spaces.
  - System, User and “Hardware” threads

---

**Virtual Registers**

- Per-thread “registers” defined by the microkernel
- Are realised via real machine registers or via memory locations
- Realisation depends on architecture and ABI
- Two basic types (was three)
  - Thread Control Registers (TCRs)
    - Used to share information about threads between the kernel and user-level
  - Message Registers (MRs)
    - Used to send messages between threads. Contains the message (or description of it, e.g. region of memory)
  - Buffer Registers (BRs)
    - Used to specify where messages (other than MRs themselves) are received

---

**Thread Control Blocks (TCBs)**

- State of a thread is stored in it’s TCB
- Some sensitive state can only be modified via a controlled interface (system calls)
  - e.g. address space associated with the thread
  - e.g. kernel stack pointer
- Some state can be freely visible and modifiable by user-level applications without compromising the system
  - Kept in user-level TCB (UTCB)
  - Include virtual registers not bound to real register
  - must only be modified via the provided library function!
  - No consistency guarantees otherwise
  - Library function scope limited to current thread
  - Many fields only modified as a side effect of some operations (IPC)

---

**KTCBs and UTCBs**

- Efficient and secure access to state traditionally kept in an in-kernel TCB
User-Level TCB (ARM)

UTCB Programming

- Only modified via provided programming interface
  - Don’t access it directly
- You can mostly ignore its contents
  - Most stuff is set/read in the context of other actions (e.g., IPC)
  - Not needed for project
    - E.g., processor number

Thread Identifiers

- Global Identifiers
  - Identify a thread uniquely within the system
  - Defined by the root task at thread creation
    - according to some policy
    - Note: version[5..0] = 0
- Local Identifiers
  - Identify a thread within an address space
    - Only unique and usable within an address space
    - Used for some optimizations
    - Typically the address of the thread’s UTCB
  - Note: V4 local thread IDs removed

ThreadControl()

- Create, destroy, and modify threads
- Determines thread attributes:
  - Global thread identifier
  - address space
  - The thread permitted to control scheduling parameters
    - This in known as target threads “scheduler”
    - Note: the “scheduler” does not perform CPU scheduling
  - The page fault handler “pager”
  - Location of the UTCB within the address spaces allotted UTCB area (See SpaceControl later)
- Note: ARM, the UTCB address is defined by the kernel, not ThreadControl()

ThreadControl()

- Threads can be created active or inactive.
  - Thread is active iff it has a pager
  - Create of inactive threads is used to
    - Create and manipulate new address spaces
    - Allocate new threads to an existing address space
  - Inactive threads can be activated in one of two ways
    - by privileged thread using ThreadControl()
    - By a local thread (same address space) using ExchangeRegisters()

Task

- L4 does not define the concept of a “task”
- We use it informally to mean:
  - An address space
    - UTCB area
    - Kernel interface page
    - Redirector
  - Set of threads inside that address space
    - Global thread ID
    - UTCB location
    - IP, SP
    - Pager
    - Scheduler
    - Exception handler
  - Code, data, stack(s) mapped into address space
Steps in Creating a New “Task”

1. Create an inactive thread in a new address space (AS)
   - An AS is referred to via the ID of one of its threads

   ```c
   L4_ThreadControl (task, /* new tid */
   task, /* new space identifier */
   me, /* scheduler of new thread */
   L4_nilthread, /* pager = nil, inactive, */
   (void *) -1); /* NOP Utcb location */
   ...
   ```

   ...creates a new thread in an otherwise empty address space

Steps in Creating a New “Task”

2. Define location of KIP and UTCB area in new address space

   ```c
   L4_SpaceControl (task, /* new TID */
   0, /* control */
   kip_area, /* where KIP is mapped */
   utcb_area, /* location of UTCB array */
   L4_anythread, /* no redirector */
   &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.

   ```c
   L4_ThreadControl (task, task, me,
   pager, /* new pager */
   (void *) utcb_base); /* utcb location */
   ```

   This results in the thread immediately waiting for an IPC containing the IP and SP of the new thread.

Steps in Creating a New “Task”

4. Send an IPC to the new thread with the IP and SP in the first two words of the message.

   This results in the thread starting at the received IP with the SP set as received.

Adding Threads To a Task

- Use `ThreadControl()` to assign new threads to AS.

  ```c
  L4_ThreadControl (newtid, /* new thread id */
  ExistingId, /* address space identifier */
  me, /* scheduler of new thread */
  pager, /* new pager */
  (void *) utcb_base);
  ```

- Can also create new thread inactive
  - Task can then manage new threads itself
  - Using `ExchangeRegisters()`

Practical Considerations

- Above sequence for creating tasks and threads is cumbersome
  - Price paid for leaving policy out of kernel
  - Any shortcuts imply policy

- A system built on top of L4 will inherently define policies
  - Can define and implement library interface for task and thread creation
  - Incorporate system policy
  - See `sos_task_new()` for an example

- Actual apps would not use raw L4 system calls, but
  - Use libraries
  - Use IDL compilers
Manipulating threads within an Address Space

- So far can
  - Create a new address space with a single thread
  - Assign new threads to an existing address space
- ExchangeRegisters
  - Used to activate or modify an existing thread within an address space.

**ExchangeRegisters()**

\[
\text{L4_ThreadId_t L4_ExchangeRegisters (L4_ThreadId_t dest, L4_Word_t control, L4_Word_t sp, L4_Word_t ip, L4_Word_t flags, L4_Word_t UserDefHandle, L4_ThreadId_t pager, L4_Word_t *old_control, L4_Word_t *old_sp, L4_Word_t *old_ip, L4_Word_t *old_flags, L4_Word_t *old_UserDefHandle, L4_ThreadId_t *old_pager)}
\]

**Threads**

- Note the microkernel only manages (preserves) the user-level IP and SP
  - (and registers if preempted)
- The following is managed by user-level applications (This means you)
  - User stack location, allocation, size, deallocation
  - Thread ID allocation, deallocation
  - Entry point
  - UTCB slot allocation
    - KIP specifies UTCB requirements

**Be CAREFUL!!!!!**

- Stack corruption is a common problem
- Stack corruption is very difficult to
  - diagnose
  - debug
Communication

IPC Overview

- Single system call that implements synchronous unbuffered IPC
  - Has a send and receive component
    - Either can be omitted
  - Receive operation can
    - specify a specific thread from which to receive ("closed receive")
    - specify willingness to receive from any thread ("open wait")
      - can be any thread in system or any local thread (same AS)
  - Results in five different logical operations:
    - Send() send a message to a specified thread
    - Receive() "closed" receive from a specific sender
    - Wait() "open" receive from any sender
    - Call() send and wait for reply
      - typical client RPC operation
    - Reply_and_Wait() send to a specific thread and wait for any new message
      - typical server operation

Thread Identifiers

- Global Identifiers
  - Thread IDs
  - Interrupt IDs
  - Special IDs
    - Nil thread
    - Any thread

Thread No (32)  
Version (32)  
Interrupt No (32)  
1 (32)  
0 (64)  
-1 (64)

In <l4/types.h>

#define L4_nilthread ((L4_ThreadId_t) { raw : 0UL})
#define L4_anythread ((L4_ThreadId_t) { raw : ~0UL})
#define L4_anylocalthread ((L4_ThreadId_t) { local : { X : {L4_SHUFFLE2(0, ~0UL)}}})

IPC Registers

- Message registers
  - virtual registers
    - not necessarily hardware registers
    - part of thread state
  - on ARM 6 physical registers, rest in UTCB
    - actual number is system-configuration parameter
  - at least 8, no more than 64
  - contents form message
    - first is message tag, defining message size (etc)
    - rest untyped words, not (normally) interpreted by kernel
    - kernel protocols define semantics in some cases
  - Simple IPC just copies data from sender’s to receiver’s MRs!
    - this case is highly optimised in the kernel ("fast path")
    - Note: no page faults possible during transfer (registers don’t fault!)

Message Register IPC

Message transferred from one thread’s MRs to the other thread’s MRs

Guaranteed to not cause page faults
Overview of IPC operations

- L4_Ipc system call performs all IPC operations (both sending and receiving)
- Helper functions for frequent operations (see <l4/ipc.h>)
  - L4_Send
    - Send a message to a thread (blocking)
  - L4_Receive
    - Receive a message from a specified thread
  - L4_Wait
    - Receive a message from any sender
  - L4_Call
    - Send a message to a particular thread and wait for it to respond (usual RPC operation)

Example: Sending 4 untyped words

- Only 5 MRs transferred
  - Note: On ARM, 6 MRs are transferred in CPU registers
  - Fast
  - The rest (if used) are copied from and to memory

Note: u, r and t are implicitly by L4_MsgAppendWord()
Note: Ideally we would use an IDL compiler instead of generating message manually

Example IPC code

```c
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_MsgLoad(&msg);
tag = L4_Send(tid);
```

IPC result MR0

- MsgTag [MR0]
  - u untyped words received (u = 0, send only IPC)
  - v typed words sent/received (v = 0, send only IPC)
  - Flags E|X|p
    - E: error occurred (send or receive), see ErrorCodeTCR for details
    - X: received cross processor IPC (ignore)
    - p: received redirected IPC (later)

- MR0
  - Specifies message content
  - u: number of words in message (excluding MR0)
  - p: specifies propagation
    - allows sending a message on behalf of another thread
      - specified by virtual sender in UTCB
      - receiver gets from kernel virtual, rather than real sender ID
      - restricted for security (essentially allowed for local threads)
  - r: specifies asynchronous notification operation (later)
  - blocking receive
    - if unset, fail immediately if no pending message
  - s: blocking send
    - if unset, fail immediately if receiver not waiting
  - label: user-defined (e.g., opcode)
### IPC

#### Send
- **dest**
- **nilthread**
- **MR**

#### Receive
- **nilthread**
- **MR**

#### Wait
- Receive from anyone
- **nilthread**
- **anythread**

#### Call
- **dest**
- **MR**

#### ReplyWait
- **dest**
- **next**

#### Obsolete IPC features
- Currently, StringItems are not supported on the MIPS-64 and ARM version of L4Ka::Pistachio
- Alternatives
  - Break long messages into many short messages
  - Share memory
- Map and Grant "typed" items in IPC not supported
  - used to send page mapping via IPC
  - use MapControl() system call instead
- Timeout on IPC
  - limit blocking time
  - limited use in practice
  - replace by send/receive block bits (s,r).
Drivers at User Level

- IO ports: part of the user address space
- Interrupts: messages from "hardware" threads
- Acknowledge hardware interrupt via replying to interrupt message

Interrupts

- Interrupts: messages from "hardware" threads
- Acknowledge hardware interrupt via replying to interrupt message

Interrupt Handlers

- Typical setup: interrupt handler is bottom-half device driver
- Interrupt handling:
  1. interrupt is triggered, hardware disables interrupt and invokes kernel
  2. kernel masks interrupt, enables interrupts and sends message to handler
  3. handler receives message, identifies interrupt cause, replies to kernel
  4. kernel acknowledges interrupt
  5. handler queues request to top-half driver, sends notification to top half, waits for next interrupt

Interrupt Association

- Association is done via the privileged thread (root task) using ThreadControl.
  - To associate a thread to an interrupt
    - Set the pager of the hardware thread ID to the thread ID of the interrupt handler
  - To disassociate the thread from an interrupt
    - Set the pager of the hardware thread ID to the hardware thread ID itself
INTERRUPTS

Modelled as IPC messages sent by virtual hardware threads
- received by interrupt handler thread registered for that interrupt
- empty (MR 0) reply to interrupt thread acknowledges interrupt

Interrupt handler association is via ThreadControl()
- set the hardware thread’s pager to the handler thread
- dissociate by setting the pager to the hardware thread’s own ID

COMP9442 06/S2 L4 PROGRAMMING

INTERRUPT HANDLES

System Calls

KernelInterface ✔
ThreadControl ✔
ExchangeRegisters ✔
IPC ➜ ThreadSwitch

Schedule ✔
MapControl ✔
SpaceControl ✔
ProcessorControl ✔
CacheControl ✔

ThreadSwitch()

Forfeits the caller’s remaining time slice
- Can donate remaining time slice to specific thread
  - that thread will execute to the end of the time slice on the
  - donor’s priority
- Can donate remaining time slice to specific thread
  - that thread will execute to the end of the time slice on the
  - donor’s priority

Directed donation can be made for
- can be made immediately
- command line option
- current interrupting thread
- command line option
- if no recipient specified (or recipient is not runnable)
  - the next time the context is ready
- directed donation is only valid to
  - the thread then executing

Directed donation can be used for
- explicit scheduling of threads
- implementing wait-free locks
- ...

ThreadSwitch()

Directed donation can be made for
- can be made immediately
- command line option
- current interrupting thread
- command line option
- if no recipient specified (or recipient is not runnable)
  - the next time the context is ready
- directed donation is only valid to
  - the thread then executing

Directed donation can be made for
- can be made immediately
- command line option
- current interrupting thread
- command line option
- if no recipient specified (or recipient is not runnable)
  - the next time the context is ready
- directed donation is only valid to
  - the thread then executing

COMP9442 06/S2 L4 PROGRAMMING
SCHEDULE

- The caller may invoke a scheduler.
- Nor does it actually schedule any threads.
- CALL: 
  • Returns (expensive) scheduler invocation frequent
  • $\text{IPC}$ can be used to implement lock-free synchronization
    - except when the caller intends to execute the operation
    - for $\text{IPC}$ a scheduled (not delayed) thread
      - When the scheduler sends a thread to the kernel, the thread
        - may be scheduled again
- $\text{IPC}$ may involve a scheduler
- Each thread has:
  - a priority, determines whether it is scheduled
  - a time slice length, determines, once scheduled, when it will
    be preempted
  - a total quantum
  - When scheduled, the thread gets a new time slice
    - the time slice is subtracted from the thread's total quantum
    - when total quantum is exhausted, the thread's scheduler is notified
      - can change
- The scheduler may defer preemption:
  - unless there is a runnable thread of higher than the
    sensitive priority
  - for up to a specified maximum delay
  - exceeding this causes an IPC to the exception handler
  - can be used to implement lock-free synchronization

TOTAL QUANTUM AND PREEMPTION IPC

- The /CB/CW/CT/CS/D9/D0/CT/B4/B5 syscall does not
  invoke a scheduler!
- Nor does it actually schedule any threads.
- /CB/CW/CT/CS/D9/D0/CT/B4/B5 manipulates a thread's scheduling parameters:
  - The caller must be registered as the destination's scheduler.
- The caller may invoke a scheduler.
- The scheduler may defer preemption:
  - unless there is a runnable thread of higher than the
    sensitive priority
  - for up to a specified maximum delay
  - exceeding this causes an IPC to the exception handler
  - can be used to implement lock-free synchronization

SYSTEM CALLS

- KernelInterface ✔
- ThreadControl ✔
- ExchangeRegisters ✔
- IPC ✔
- ThreadSwitch ✔
Address Spaces

- Special Pages:
  - Table of page types
  - Used to specify mappings of objects

Address Spaces:

- Address spaces are a second-class abstraction
- Can also be done proactively

Address Spaces:

- Need to be explicitly populated with page mappings
- Address spaces are created empty

Address Spaces:

- On ARM, n ≥ 12

Address Spaces:

-sizes (6)

Address Spaces:

-sizes (base/1024)

Address Spaces:

-On ARM, s ≥ 12

Address Spaces:

- Special Page Types:
  - Page of size n ≥ 2
  - Page is a special page
  - Page of size ≥ 2
  - Page is a special page
  - Page is a special page
  - Page is a special page
  - ...
### MAP ITEM

- Specifies a mapping to be created in destination AS

**fpage**
- Specifies where mapping is to occur in destination AS

**phys adr**
- Base of physical frame(s) to be mapped

**Note:** shifted 4 bits to support 64MB of physical AS

**attr**
- Memory attributes (e.g., cached/uncached)

**rxw**
- Permissions

**access rights in destination address space**
- Can be used to change (up/downgrade) rights

Removing all rights removes the mapping (unmap operation)

### PAGE FAULT HANDLING

1. **Application**
   - kernel receives request from application
   - generates page fault message

2. **Page**
   - map item for map 4kB page at P A 0xc0000:
     - Phys addr must be aligned to page size
     - E.g., page headers were page fault at 0x2000:

3. **Page Fault Message**
   - Format of kernel-generated page fault message
     - Fault IP
     - Fault address
     - Permissions
     - Memory attributes

4. **Pager**
   - E.g., pager handles write page fault at 0x2002
     - map item for map 4kB page at P A 0xc0000:
     - note: Phys addr must be aligned to page size
      
### PAGE ACTION

1. Application sends a page fault message to pager
2. Pager establishes a mapping to the page in the address space
3. Pager replies to the page fault message
   - Content of message ignored
   - Only serves as a synchronization signal
4. Application receives a mapping to be created in destination AS
ProcessorControl

- Sets processor core voltage and frequency (where supported)

System Calls

- ProcessorControl
- SpaceControl
- MemoryControl
- ThreadControl
- ExchangeRegisters
- Schedule
- IPC
- ExchangeRegisters
- MemoryControl
- KernelInterface
CACHE CONTROL

- Privileged system call
- Some functions can be called from anywhere (Hack!)
  - Privilege check
  - Address range in global memory
  - Kind of operation (read/write/lock/unlock)
  - Target cache (L1/L2/L3)
  - Flush to flush caches or lock cache lines as per arguments

That's it!
Each thread has three scheduling attributes:

- **Priority**
- **Time slice length**
- **Total quantum**

Kernel schedules runnable threads according to their priority:

- Round-robin between threads of equal priority.

When a thread is scheduled:

- It is given a fresh time slice.
- The time slice is deducted from its total quantum.

When total quantum is exhausted, the kernel sends a message on behalf of the preempted thread to its scheduler:

- The scheduler can provide a new quantum using the following commands:
  - `/CB
/CW/CT/CS/D9/D0/CT/B4/B5`

An exception may trigger with modified thread state:

- Exception handler may reply with modified thread state:
  - `retry`:
    - Reply with unchanged state.
    - Possibly after removing cause.
    - Possibly changing other parts of state (registers).
  - `continue`:
    - Reply with appropriate registers.
    - (assuming 4-byte instructions)
  - `emulation`:
    - Compute desired result.
    - Reply with appropriate register value and
      `/C1/C8/B7/BP/BG`.
  - `handler`:
    - Reply with IP of local exception handler code to be executed by the thread itself.
  - `ignore`:
    - Will block the thread indefinitely.
  - `kill`:
    - Use `/BX/DC

Other exceptions, not handled by the thread's exception handler:

- `Page fault`
- `Thread start`
- `Interrupt`
- `Preemption`
- `Exception`
ASYNCHRONOUS NOTIFICATION

• Very restricted form of asynchronous IPC:
  ➜ delivered without blocking sender
  ➜ delivered immediately, directly to receiver's AS
  ➜ message consists of a bit mask OR-ed to receiver's bitfield

• Two ways to receive asynchronous notifications:
  ➜ synchronously by a form of blocking IPC wait
    ➜ receiver specifies mask of notification bits to wait for
    ➜ on notification, kernel manufactures a message in a defined format
  ➜ asynchronously by checking in UTCB
    ➜ but remember it's asynchronous and can change at any time!