L4 Programming Introduction

**Fundamental Concepts**

- **Address Spaces**
  - Unit of protection, resource management
- **Threads**
  - Execution abstraction and provide unique identifiers
  - Communication: IPC
    - Synchronous
    - Identification: uids
  - AS construction: mapping
    - Via IPC
    - Flexpages
      - Architecture independent page abstraction

**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

**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 version and configuration of the kernel and the machine it’s running on
  - Examples: Page sizes supported, API version, physical memory layout

**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.

**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)
  ```
Virtual Registers

- Per-thread “registers” defined by the microkernel
- Are realised via real machine registers or via memory locations
  - Realisation depended on architecture and ABI
- Three basic types
  - 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

Threads

- Code, data
  - Code
  - Data

Traditional Thread

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

L4 thread = trad. thread +

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

Thread Control Blocks (TCBs)

- State of a thread is stored in it’s thread control block
- Some state can only be modified via a controlled interface (system calls) (e.g. address space associated with the thread)
- Some state can be freely visible and modifiable by user-level applications without compromising the system
- Why not put this information in a user-level TCB (UTCB) for efficiency of access

Thread Control Registers

- Stored in UTCB
- 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

- Processor
- UserDefinedHandle (rw, Threads)
- ExceptionHandler (rw)
- XferTimeouts (rw, IPC)
- ErrorCode (ro, IPC)
- Virtual/ActualSender (rw, IPC)
Thread Identifiers

- Global Identifiers
  - Identify a thread uniquely within the system
  - Thread number and version number assigned by root task accord to whatever policy you like
  - Example:
    - Version numbers are unique to give unique Ids
    - Threads number are grouped into tasks to allow upper bits to be task numbers
- Local Identifiers
  - Assigned by the system
  - Assigned by the system

Thread Control

- Used to create, destroy, and modify threads
- Determines:
  - The global thread identifier associated with the thread
  - The address space the thread is associated with
  - The thread permitted to control scheduling parameters of the new thread
  - The pager
  - Location of the UTCB within the address spaces allotted UTCB area (See SpaceControl later)
- Threads can be created active or inactive.
  - Inactive is used to create and manipulate a new address space, or allocate a new thread to an existing address space.

Steps in Creating a New "Task"

1. Create an inactive thread in a new address space
   - L4_ThreadControl (task, /* new tid */
   - task, /* new space identifier */
   - task, /* scheduler of new thread */
   - L4_nilthread, /* pager = nil, inactive, */
   - L4_nilthread, /* scheduler of new thread */
   - L4_nilthread, /* pager = nil, inactive, */
   - (void *) -1); /* NOP Utcb location */
Steps in Creating a New “Task”

2. Initialise location of KIP and UTCB area in new address space
   L4_SpaceControl (task,
   0, /* control (ignored on mips) */
   kip_area,
   utcb_area,
   L4_anythread, /* redirector */
   &control);

Steps in Creating a New “Task”

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

This is a little cumbersome!

• We provide the following support function in the sample project code.
  – Read it and understand what it does!!!!
  l4e_task_new(L4_ThreadId_t task,
             L4_ThreadId_t pager,
             void *entrypoint, void *stack)

Adding extra inactive threads to a task

• Use ThreadControl to assign new inactive threads to an existing address space.
  • L4_ThreadControl (newtid, /* new thread id */
                         ExistingId, /* address space identifier */
                         me, /* scheduler of new thread */
                         L4_nilthread, /* pager = nil, inactive,*/
                         (void *) -1); /* NOP Utcb location */
  • Note: Can also add active threads

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

L4_ThreadId_t L4_ExchangeRegisters (L4_ThreadId_t dest,
L4_Word_t control,
L4_Word_t sp,
L4_Word_t ip,
/* ignore */ 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,
/* ignore */ 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 allocation, deallocation
  - Entry point

Be CAREFUL!!!!

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

IPC Overview

- Single system call that implements several variants of synchronous unbuffered IPC
  - Arguments determine IPC system call behaviour
  - Operations are
    - Send() send a message to a specified thread
    - Receive() "closed" receive from a specific sender (might be an interrupt)
    - Wait() "open" receive from any sender (incl. interrupt).
    - Call() send and wait for reply (usual RPC operation)
    - Reply_and_Wait() send to a specific thread and wait for any new message (typical server operation)

Communication

Ignoring Address Spaces
Thread Identifiers

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

In `<l4/types.h>`

```c
typedef union {
  L4_Word_t raw;
  struct {
    L4_BITFIELD2(L4_Word_t,
      version : __14,
      thread_no : __18);
  } X;
} L4_GthreadId_t;

typedef union {
  L4_Word_t raw;
  struct {
    L4_BITFIELD2(L4_Word_t,
      __zeros : 6,
      local_id : 26 __PLUS32);
  } X;
} L4_LthreadId_t;

typedef union {
  L4_Word_t raw;
  L4_GthreadId_t global;
  L4_LthreadId_t local;
} L4_ThreadId_t;
```

IPC Registers

- Message Registers
  - 64 "registers"
  - Form a message
  - Used to transfer typed items and untyped words
    - Typed items
      - MapItem
      - GrantItem
    - StringItem
- Buffer Registers
  - 34 "registers"
  - Specify where
    - MapItems and GrantItems are received
    - StringItems are received if any are permitted to be in the message

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_ReplyWait  
    - Send a response to a thread and wait for the next message
  - L4_Call  
    - Send a message to a particular thread and wait for it to respond (usual RPC operation)
• Message content specified by MR0
  – \( u \) the number of untyped words
  – \( t \) the number of words holding typed items
  – label free for the sender to use as part of the message (usually a "label" or "tag")
  – flags specifies option for the IPC operation
    • Not used for project (set = 0)
    • Specifies propagation

Example: Sending 4 untyped words

• Only 5 MRs transferred
  – Note: On MIPS64, 9 MRs are transferred in CPU registers
    • Fast
    • The rest (if used) are copied from and to memory

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)
  – \( t \) typed words sent/received (\( t = 0 \), send only IPC)
  – Flags EXrp
    – E: error occurred (send or receive), see ErrorCode TCR for details
    – X: received cross processor IPC (ignore)
    – r: received redirected IPC (ignore)
    – p: received propagated IPC (ignore)

The StringItem Type

• Example sends a single simple string + two untyped words
  – More complex variations are possible (see the reference manual)
• Used to send a message in place
  – Avoid marshaling costs

Receiving Strings

• Buffer Registers used to specify area and size of memory region to receive strings
• Simple example
  – A single string

Note: The typed items always follow the untyped words
C: specifies if typed items follow
Note!!!!

- Currently, StringItems are not supported on the MIPS-64 version of L4Ka::Pistachio
- We will discuss alternatives later
  - Example: Break long messages into many short messages

IPC Timeouts

- Used to control the duration of IPC
- Two timeout types
  - Rcv/Snd Timeouts
    - Used to control how long the IPC syscall will block prior to
      - The send phase beginning (SndTimeout)
      - The receive phase beginning (RcvTimeout)
  - XferTimeouts (Snd/Rcv)
    - Used to limit how long the IPC transfer takes
      - Only used for StringItems (i.e. you can ignore them on MIPS-64)
      - Needed to limit time waiting for sender/receiver pagefaults on memory.
        » More later in the course

Timeouts

- Timeouts
  - snd timeout, rcv timeout, xfer timeout

Timeouts

- Timeouts
  - snd timeout, rcv timeout, xfer timeout

relative timeout values

- 0
- infinite
- 1us ... 610h (log)
- 27.1ms
## Timeout Value Range

<table>
<thead>
<tr>
<th>m</th>
<th>e</th>
<th>m = 10^23</th>
<th>0</th>
<th>1.00E-06</th>
<th>1.02E-03</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.00E-06</td>
<td>2.09E-03</td>
<td>2</td>
<td>2.06E-01</td>
<td>2.68E+02</td>
</tr>
<tr>
<td>2</td>
<td>4.00E-06</td>
<td>4.18E-03</td>
<td>3</td>
<td>4.06E-01</td>
<td>5.38E+03</td>
</tr>
<tr>
<td>3</td>
<td>8.00E-06</td>
<td>8.36E-03</td>
<td>4</td>
<td>8.00E-01</td>
<td>1.07E+03</td>
</tr>
<tr>
<td>4</td>
<td>1.60E-05</td>
<td>1.64E-02</td>
<td>5</td>
<td>1.58E+00</td>
<td>2.19E+03</td>
</tr>
<tr>
<td>5</td>
<td>3.20E-05</td>
<td>3.27E-02</td>
<td>6</td>
<td>2.15E+00</td>
<td>4.19E+03</td>
</tr>
<tr>
<td>6</td>
<td>6.40E-05</td>
<td>6.55E-02</td>
<td>7</td>
<td>4.10E+00</td>
<td>8.38E+03</td>
</tr>
<tr>
<td>7</td>
<td>1.28E-04</td>
<td>1.21E-01</td>
<td>8</td>
<td>8.19E+00</td>
<td>1.68E+01</td>
</tr>
<tr>
<td>8</td>
<td>2.56E-04</td>
<td>2.42E-01</td>
<td>9</td>
<td>1.28E+01</td>
<td>1.35E+02</td>
</tr>
<tr>
<td>9</td>
<td>5.12E-04</td>
<td>5.24E-01</td>
<td>10</td>
<td>2.56E+01</td>
<td>2.65E+02</td>
</tr>
<tr>
<td>10</td>
<td>1.02E-03</td>
<td>1.05E+00</td>
<td>11</td>
<td>5.12E+01</td>
<td>5.32E+02</td>
</tr>
<tr>
<td>11</td>
<td>2.05E-03</td>
<td>2.09E+00</td>
<td>12</td>
<td>1.04E+02</td>
<td>1.05E+03</td>
</tr>
<tr>
<td>12</td>
<td>4.10E-03</td>
<td>4.18E+00</td>
<td>13</td>
<td>2.09E+02</td>
<td>2.15E+03</td>
</tr>
<tr>
<td>13</td>
<td>8.20E-03</td>
<td>8.36E+00</td>
<td>14</td>
<td>4.18E+02</td>
<td>4.29E+03</td>
</tr>
<tr>
<td>14</td>
<td>1.64E-02</td>
<td>1.64E+01</td>
<td>15</td>
<td>8.36E+02</td>
<td>8.58E+03</td>
</tr>
<tr>
<td>15</td>
<td>3.28E-02</td>
<td>3.28E+01</td>
<td>16</td>
<td>1.68E+03</td>
<td>1.72E+04</td>
</tr>
</tbody>
</table>

## Timeouts

- **Timeouts**
  - snd timeout, rcv timeout, xfer timeout

  - relative timeout values
    - 0
    - infinite
    - 1us ... 610 h (log) \(2^m\) µs
  - absolute timeout values
    - clock \(m\)
    - clock + \(2^{(e+10)}\)

## IPC

- **Send**
  - to / from
  - FromSpecifier
  - Timeouts
  - MR

- **Receive**
  - from dest
  - to / from
  - FromSpecifier
  - Timeouts
  - MR

- **Wait**
  - Receive from anyone
  - to / from
  - FromSpecifier
  - Timeouts
  - MR
**IPC**

- Call
  - `dest` to/from
  - FromSpecifier
  - Timeouts
  - `MR_0`

**IPC**

- ReplyWait
  - `dest` to/from
  - Anythread
  - FromSpecifier
  - Timeouts
  - `MR_0`

**Interrupts**

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

**Interrupt Associated**

- 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

**Sample Code**

```c
int register_cpu_interrupt_handler(host_handle_t host, int irq, void *fn, void *data)
{
    L4_ThreadId_t tid;
    int res;
    tid.global.X.thread_no = irq;
    tid.global.X.version = 1;
    res = L4_ThreadControl(tid, tid, L4_nilthread, L4_Pager(), (void*) -1);
    // The tid we want
    if (res != 1) {
        l4e_printf("BADNESS ON THREAD CONTROL\n");
    }
    irq_handlers[irq].function = fn;
    irq_handlers[irq].data = data;
    return 1;
}
```

**Microkernel System Calls**

- IPC
- Unmap
- SpaceControl
- ThreadSwitch
- Schedule
- SystemClock
- ExchangeRegisters
- ThreadControl
- KernelInterface
- ProcessorControl
- MemoryControl
ThreadSwitch

- Yields the current thread’s remaining timeslice, or donate remaining timeslice to another thread
  - You should not do this, but...
    - Can use to reduce impact of busy-wait
    - Ensure progress of resource (lock) holder
- L4_ThreadSwitch (thread);
  - Thread = nilthread: Yield the processor
  - Thread = threadID: Donate timeslice
  - See <l4/schedule.h> for derived functions

Schedule

- L4 implements a mostly multi-level round robin scheduler
- Schedule is used:
  - to change scheduling parameters of threads (which you should not need to do).
    - Timeslice
    - Priority
    - Total quantum (don’t use, set to infinity)
  - for controlling preemption parameters: not implemented
  - set the processor the thread should run on: not needed, you have only one 😌

SystemClock

- Returns the current system clock
  - 64-bit number that counts μ-seconds
  - Usually not a real system call

Microkernel System Calls

- IPC
- Unmap
- SpaceControl
- ThreadSwitch
- Schedule
- SystemClock
- ExchangeRegisters
- ThreadControl
- KernelInterface
- ProcessorControl
- MemoryControl

Address Spaces
Address Spaces

- map
- unmap
- grant

Page Fault Handling

Application → Page

"PF" msg

map msg

Application → Page

"PF" msg

map msg
Address Spaces

Address Space Management

Page Fault Protocol

PageFault Message

Mapping Questions

Fpage Data Type

- Note: Applications can synthesis page fault messages
  - Not a problem as the application could do it anyway by directly accessing the memory it wishes to cause a fault on

- Fpage
  - fpage size = 2^s
  - Specifies a region of memory that is
    - A power of 2 in size
    - Aligned to its size
  - Note: Smallest supported size is architecture specific
    - MIPS-64 supports 4K (s = 12)
Fpage Data Type

- Complete Address Space
  - Nilpage

Buffer Register 0

- Specifies
  - Willingness and locations to receive StringItems
    - \( s = 1 \)
  - The receive window for mappings
    - Which location in the address space mappings are allowed

MapItem/GrantItem Data Type

- Permissions
  - R: read
  - w: write (1) mapping
  - X: execute
  - Note: Not all architectures support all combinations
    - MIPS-64: rx, rwx are supported by hardware
  - g: mapping (0) or granting (1)
Receive window > mapping size

Mappings and Window Sizes
- See reference manual for precise definition of what happens for mismatched mappings and window sizes
- Advice:
  - Simply use 4K pages for all mappings

A Map Message
- For page faults, the kernel expects the following map message response
  - No untyped words
  - 1 MapItem

Explicit Mapping Receive
- BR₀ determines whether a receive/wait IPC can include a mapping
  - Set it prior to invoking IPC receive/wait

Unmap
- Used to revoke a mappings established in other address spaces that are derived from mappings in the current address space
- Unmap is also used to revoke access rights to existing mappings
  - Example: RW → RO
- The mapping to revoke are specified by fpages in MRs

Unmap Arguments
- Control
  - f: specifies whether fpage is flushed from the current address space in addition to revoking derived mapping
  - k: specifies the highest number MR that contain an Fpage to unmap
- Fpages
  - Fpages specify the regions in the local address space
  - rw: the access rights to revoke
    - The mapping to revoke are specified by fpages in MRs
### Unmap Results

- **RWX**
  - Reference (r), Dirty (w), and Executed (x) bits
  - Reset as a result of the unmap operation
  - Bit returned set if corresponding access has occurred on any derived mapping

- Note: Should not need to use
  - Behaviour is not heavily tested

### SpaceControl

- Used to control the layout of newly created address spaces
  - Specifically
    - Location of Kernel Info Page
    - Location of UTCB region
  - Redirector
    - All IPC from threads within the address space is redirected to a controlling thread
    - Used to enforce security policy
  - Note: Should not need to change what is already done in the example code

### ProcessorControl

- Privileged system call
- Sets processor frequency, voltage and other processor specific stuff.
- You can safely ignore it

### Microkernel System Calls

- IPC
- Unmap
- SpaceControl
- ThreadSwitch
- Schedule
- SystemClock
- ExchangeRegisters
- ThreadControl
- KernelInterface
- ProcessorControl
- MemoryControl

### MemoryControl

- Privileged system call
- Used to set cache architecture attributes on pages in memory
  - Machine specific
  - See Appendix E for details

### Microkernel System Calls

- IPC
- Unmap
- SpaceControl
- ThreadSwitch
- Schedule
- SystemClock
- ExchangeRegisters
- ThreadControl
- KernelInterface
- ProcessorControl
- MemoryControl

That's all folks
Protocols

- Page Fault
- Thread Start
- Interrupt
- Preemption
  - Not supported, used to control preemption
- Exception
- Sigma0

Exception Protocol

- Exception include: Divide by Zero, etc.
- Exceptions are indicated via Exception IPC to the thread’s exception handler thread
  - The IPC contains
    - IP of where to resume the thread after handling the exception
    - Exception type
    - Other machine specific stuff
  - The exception handler can respond with an IPC specifying a new IP and other state to recover from the exception
- You should not need to do anything other than kill the task that caused the exception

Sigma 0

- Contains all physical memory in the machine
  - Except that reserved for kernel use
  - Mapped idempotently
    - One-to-one
- Sigma0 distributes physical memory to start-up tasks at boot time
  - It maps each page once (and only once)
- Initial tasks request memory via an IPC protocol that allows mappings to be received
- Sigma0 responds (if possible) with an idempotent mapping giving access to a frame of physical memory

Sigma0 Request Message

- Requested attributes
  - Architecture specific
    - Use default = 0
  - Requested Fpage
    - B != -1
    - Request a specific region of physical memory

Sigma0 Response

- If a successful request, Sigma0 responds with an idempotent mapping giving access to the physical memory request
Sigma0 Response

- If a unsuccessful request, Sigma0 responds with the following

<table>
<thead>
<tr>
<th>ϕ</th>
<th>R₀</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>φ</td>
</tr>
</tbody>
</table>