L4 Programming

COMP9242
2006/S2 Week 2

Recap: L4 Abstractions and Mechanisms

Three basic abstractions:
- Address spaces
- Threads
- Time (second-class abstraction)

Two basic mechanisms:
- Inter-process communication (IPC)
- Mapping

L4 API:
- 10 system calls (N2, other APIs have slightly different numbers)
- 6–8 kernel-defined protocols

L4 System Calls

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

Kernel Interface Page (KIP)

- Kernel memory object
  - mapped into address space (AS) at creation time
  - location defined by /sys/kernel/kmem
  - KernelInterface() syscall returns address
- Contains information about kernel and hardware
  - kernel version
  - supported features (page sizes)
  - physical memory layout
  - system call addresses
- C language API
  

L4_KERNEL_INTERFACE (l4_Void_t *ApVersion,
  l4_Void_t *ApFlags,
  l4_Void_t *KernId)
**System Calls**

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

**Threads**

- Traditional thread:
  - execution abstraction
  - consists of:
    - registers (GP and status registers)
    - stack
- L4 thread also has:
  - **virtual registers**
  - scheduling priority and time slice
  - unique thread-ID
  - address space
- L4 provides for a fixed overall number of threads
  - system, user and “hardware” threads
  - user threads created/deleted/allocated by privileged root task

**Virtual Registers**

- Kernel-defined, user-visible thread state
- Implemented as physical machine registers or memory locations
  - depends on architecture and ABI
- Three types
  - **thread control registers (TCRs)**
    - for sharing info between kernel and user
  - **Message Registers (MRs)**
    - contain the message passed in an IPC operation

**Thread Control Block (TCB)**

- Contains thread state
  - kernel-controlled state, must only be modified by syscalls
    - kept in kernel TCB (KTCB)
  - state that can be exposed to user w/o compromising security
    - kept in user-level TCB (UTCB)
    - includes virtual registers (as far as not bound to real registers)
    - must only be modified via the provided library functions!
    - No consistency guarantees otherwise
    - many fields only modified as side effect of some operations (IPC)
**User-Level TCB (ARM)**

- PreemptedIP (32)
- PreemptCallbackIP (32)
- VirtualSender/ActualSender (32)
- IntendedReceiver (32)
- ErrorCode (32)
- ProcessorNo (32)
- NotifyBits (32)
- NotifyMask (32)
- Acceptor (32)
- ~ (16)
- cop (8)
- pmpt (8)
- ExceptionHandler (32)
- Pager (32)
- UserDefinedHandle (32)
- MyGlobalId (32)

**Global Thread ID**

- thread_no (16)
- version (14)

**Global Interrupt ID**

- interrupt_no (16)
- t (16)

**Thread Identifiers**

- Global thread IDs
  - uniquely identify a thread system-wide
  - defined by root task at thread creation
  - ... according to some policy
  - Note: version[5..0] ≠ 0
- Note: V4 local thread IDs removed

**ThreadControl()**

- Can create threads active or inactive
  - thread is active iff it has a pager
  - creation of inactive threads is used to
    - create and manipulate new address spaces
    - allocate new threads to existing address spaces
  - inactive threads can be activated in one of two ways
    - by a privileged thread using threadControl()
    - by a local thread (same address space) using
      ExchangeRegisters()

**ARM: utcb must be zero!**
### TASK

- L4 does not define a concept of a "task"
- We use it informally meaning:
  - 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

### CREATING A TASK

1. Create inactive thread in a new address space (AS)
   - Note: L4 does not (presently) support first-class names for AS!
   - An AS is referred to via the ID of one of its threads

   ```c
   L4_ThreadID_t task - according to policy;
   L4_ThreadID_t me = L4_myself();
   L4_ThreadControl (task, /* new TID */
    task, /* new address space */
    me, /* scheduler of new thread */
    L4_sithread, /* pager, nh=inactive */
    (void*)-1); /* no utcb yet */
   ```

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

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

   ```c
   L4lparr osControl (task, /* new TID */
    0, /* control */
    kip_page, /* where KIP is mapped */
    utc_page, /* location of UTCB array */
    l4낯thread, /* no redirector */
    &control); /* leave alone ;-) */
   ```

3. Define UTCB address of new thread

   ```c
   utc_base = L4뗡page;
   L4_ThreadControl (task, task, me,
    pager, /* new pager */
    (void*) utc_base);
   ```

   Thread will now wait for an IPC containing IP and SP

4. Send IPC to thread containing IP, SP
   - thread will then start fetching instructions from IP

### ADDING THREADS TO A TASK

- Use `ThreadControl()` to add new threads to AS

```c
L4_ThreadID_t tid - according to policy;
 utc_base = ...;
 L4_ThreadControl (tid, task, me,
    pager, (void*) utc_base);
```

- Can create new threads inactive instead
  - task can then manage new threads itself
  - ... using `L4_setThread()`

- Note: Maximum number of threads defined at address-space creation time
  - via the size of the UTCB area
  - size and alignment conditions of UTCBs are defined in KIP
**PRÁCTICAL CONSIDERATIONS**

- Above sequence for creating tasks and threads is cumbersome
  - price to be 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 interfaces for task and thread creation
  - incorporating system policy
- Actual apps would not use raw L4 system calls, but
  - use libraries
  - use IDL compiler (Magpie)

**SYSTEM CALLS**

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

**EXCHANGE REGISTERS**

- Reads, and optionally modifies, kernel-maintained thread state

```c
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 usr_handle,
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_usr_handle,
L4_ThreadId_t old_pager)
```

- setting pager activates inactive thread
- `usr_handle` is an arbitrary user-defined value
- can be used to implement thread-local storage
- `flags` allows setting processor status bits

**EXCHANGE REGISTERS**

CPSR bits affected by `flags` (ARM):

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>N</td>
<td>negative</td>
</tr>
<tr>
<td>30</td>
<td>Z</td>
<td>zero</td>
</tr>
<tr>
<td>29</td>
<td>C</td>
<td>carry</td>
</tr>
<tr>
<td>28</td>
<td>V</td>
<td>overflow</td>
</tr>
</tbody>
</table>
**Threads and Stacks**

- Kernel does not allocate or manage stacks in any way
  - only preserves IP, SP on context switch
- User level (servers) must manage
  - stack location, allocation, size
  - entry point address
  - thread ID allocation, deallocation
  - UTCB slot allocation, deallocation
  - KIP specifies UTCB space requirements and alignment conditions
- Beware of stack overflow!

**System Calls**

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

**IPC Overview**

- Single IPC syscall incorporates a send and a receive phase
  - 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 the system, or any local thread (same AS)
- Results in five different logical operations
  - Send(): send msg to specified thread
  - Receive(): receive msg from specified thread
  - Wait(): receive msg from any thread
  - Call(): send msg to specified thread and wait for reply
  - Reply&Wait(): send msg to specified thread and wait for any message

**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 Tag MR₀

<table>
<thead>
<tr>
<th>label (16)</th>
<th>u</th>
<th>p</th>
<th>r</th>
<th>n</th>
<th>p</th>
</tr>
</thead>
</table>

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

### Example: Sending 4 Words

```c
14Msg_t msg;
14MsgTag_t tag;
14MsgClear(msg);
14MsgAppendWord(msg, word1);
14MsgAppendWord(msg, word2);
14MsgAppendWord(msg, word3);
14MsgAppendWord(msg, word4);
14Msgload(msg);
tag = 14_Send(tis);
```

Note: \( u, s, r \) set implicitly by \( 14\_MsgAppendWord \) and convenience function
Delivers \( MR₀, ..., MR₄ \) to thread \( tis \)

Note: Should use IDL compiler rather than doing this manually!

### IPC Result MR₀

<table>
<thead>
<tr>
<th>label (16)</th>
<th>u</th>
<th>p</th>
<th>r</th>
<th>n</th>
<th>p</th>
</tr>
</thead>
</table>

- Returns to receiver details of message
  - u: number of untyped words received
  - E: error occurred, check ErrorCode in UTCB
  - X: message came from another CPU
  - r: message was redirected (later)
  - p: sender used propagation, check ActualSender in UTCB

### IPC: Obsoleted Features

- String items in message
  - \( \Rightarrow \) used to send out-of-line data
  - \( \Rightarrow \) arbitrarily sized and aligned buffers
  - \( \Rightarrow \) non-essential feature that should not be in the kernel

- Map/grant items in message
  - \( \Rightarrow \) used to send page mappings through IPC
  - \( \Rightarrow \) replaced by `mapi` syscall

- Timeouts on IPC
  - \( \Rightarrow \) limit blocking time
  - \( \Rightarrow \) practically not very useful
  - \( \Rightarrow \) replaced by send/receive block bits (x, r respectively)
INTERRUPTS

- Modelled as IPC messages sent by virtual hardware threads
  - received by interrupt handler thread registered for that interrupt
  - empty (MRu=0) reply to interrupt thread acknowledges interrupt
- Interrupt handler association is via ThreadControl()
  - set the hardware thread's pager to the handler thread
  - disassociate by setting the pager to the hardware thread's own ID

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

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
  - Note: This is what the manual says.
    In the present implementation, the donation is only valid to the next timer tick (10ms on ARM)!
  - If no recipient specified (or recipient is not runnable)
    - normal "yield" operation
    - kernel invokes scheduler
    - caller might receive a new time slice immediately
- Directed donation can be used for
  - explicit scheduling of threads
  - implementing wait-free locks
  - ...
**System Calls**

- ✔ KernelInterface
- ✔ ThreadControl
- ✔ ExchangeRegisters
- ✔ IPC
- ✔ ThreadSwitch
- ➜ Schedule
  - • MapControl
  - • SpaceControl
  - • ProcessorControl
  - • CacheControl

**L4 Scheduling**

- L4 uses 256 hard priorities (0–255)
- Within each priority schedules threads round-robin
- Scheduler is invoked when
  - ➜ the current thread is preempted
  - ➜ the current thread yields
- The scheduler is not normally invoked when a thread blocks:
  - ➜ if destination thread is runnable, the kernel will switch to it
    - called direct process switch
  - ➜ scheduler only invoked if destination is blocked too
  - ➜ if both threads are runnable after IPC, the higher-prio one will run
    - presently implementation doesn’t always observe prios correctly!
- This makes (expensive) scheduler invocation infrequent

**Total Quantum and Preemption IPC**

- 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
- When the time slice is exhausted, the thread is preempted
  - ➜ preemption-control flags in the UTCB can 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 synchronisation

**Schedule()**

- The Schedule() syscall does not invoke a scheduler!
- Nor does it actually schedule any threads.
- Schedule() manipulates a thread’s scheduling parameters:
  - ➜ The caller must be registered as the destination’s scheduler
    - ➜ Set via ThreadControl()
  - ➜ can change
    - ➜ priority
    - ➜ time slice length
    - ➜ total quantum
    - ➜ sensitive priority
    - ➜ processor number
    - ➜ only relevant for SMP
    - ➜ kernel will not transparently migrate threads between CPUs
**System Calls**

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

**Address Spaces**

- Address spaces are created empty
- Need to be explicitly populated with page mappings
  - kernel does not map pages automatically (except KIP, UTCB)
- Normally AS populated by pager on demand
  - thread runs, faults on unmapped pages, pager creates mapping
- Can also be done pro-actively
  - Eg OS server can pre-map contents of executable
- Address space is a second-class abstraction
  - there are no unique identifiers for address spaces
  - an AS is identified via one of its threads (syscall TID argument)

**MapControl()**

- Creates (maps) or destroys (unmaps) page mappings
- Privileged system call (only available to root task)

```
M4_Word_t L4_MapControl (L4_Thread_t dest,
                          L4_Word_t control)
```

- dest: denounces target address space
- control: determines operation of syscall

<table>
<thead>
<tr>
<th>r</th>
<th>m</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(h)</td>
<td>(h)</td>
</tr>
</tbody>
</table>

- 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
  - map items are contained in message registers MR0, ..., MRn-1

**Specifying Mappings: Fpages**

- A ffixpage or fpage is used to specify mapping objects
  - generalisation of a hardware page
  - similar properties:
    - size is power-of-two multiple of base hardware page size
    - aligned to its size
  - fpage of size $2^s$ is specified as
    - base/1024
    - special fpages:
      - full AS
        - $0_{(h)}$ 0x0F $\sim_{(h)}$
      - null page
        - $0_{(h)}$ 0x0 0x0

- On ARM, $s \geq 12$
**Map Item**

- Specifies a mapping to be created in destination AS

<table>
<thead>
<tr>
<th>fpage (26)</th>
<th>phys adr/1024 (40)</th>
<th>attr (6)</th>
</tr>
</thead>
</table>

- 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 (eg cached/uncached)
- rwx: permissions
- access rights in destination address space
- can be used to change (up/downgrade) rights
  - (only if mapping is replaced by an otherwise identical one)
- removing all rights removes the mapping (unmap operation)

**Page Fault Handling**

- Address-spaces are populated in response to page faults
- Page faults are converted into IPC messages:
  1. app triggers page fault
  2. kernel exception handler generates IPC from faulter to pager
  3. pager establishes mapping
     - calls masstab() (if privileged) otherwise asks root task to do it
  4. pager replies to page-fault IPC
  5. kernel intercepts message, discards
  6. kernel restarts faulting thread

**Page Fault Message**

- Format of kernel-generated page fault message

<table>
<thead>
<tr>
<th>Fault IP</th>
<th>Fault address</th>
<th>MR2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0wx</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 (rx)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 (wx)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- Eg. page fault at address 0x2002: Kernel sends

<table>
<thead>
<tr>
<th>Fault IP</th>
<th>Fault address</th>
<th>MR2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x2002</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0wx</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 (rx)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- Obviously, application can manufacture same message
  - pager cannot tell the difference
  - not a problem, as application could achieve the same by forcing a fault

**Pager Action**

- E.g., pager handles write page fault at 0x2002
  - map item for map 4kB page at PA 0xc0000:

<table>
<thead>
<tr>
<th>phys</th>
<th>phys adr</th>
<th>phys</th>
<th>phys size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12</td>
<td>0x300</td>
<td>0x2000</td>
<td>0</td>
</tr>
</tbody>
</table>

- note: phys adr must be aligned to fpage size
- After establishing mapping, pager replies to page-fault message
  - content of message completely ignored
  - only servers for synchronisation: informing kernel that faulting thread can be restarted
  - if pager did not establish a suitable mapping, client will trigger same fault again
SpaceControl()

- Controls layout of new address spaces
  - KIP location (not on ARM)
  - UTCB area location (not on ARM)
- Controls setting of redirector
  - used to limit communication
  - if set to a valid thread, IPC from the AS can only be sent:
    - locally (within AS)
    - to the redirector's address space
  - any other message is instead delivered to the redirector
  - Note: not heavily tested in present version
    - your chance to pick up bonus points
- On ARM, control used to set PID register (later)

ProcessorControl()

- Sets processor core voltage and frequency (where supported)
  - used for power management
- Privileged system call
**System Calls**

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

**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 to flush from cache
- Privileged system call
  - sort-of... Some functions can be called from anywhere (Hack!)

---

**System Calls**

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

That’s it!

**L4 Protocols**

- Page fault
  - already covered
- Thread start
  - already covered
- Interrupt
  - already covered
- Preemption
- Exception
- Asynchronous notification
### Preemption Protocol

- 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 thread is scheduled
  - it is given 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
  - scheduler can provide new quantum (using `state()`)
  - not heavily tested
- Format of preemption message:

  | -3 | 0 | 0 | 0 | 0 | MR0 |

### Exception Protocol

- Other exceptions (invalid instruction, division by zero...) result in a kernel-generated IPC to thread’s exception handler
- Exception IPC
  - kernel sends (partial) thread state
    - exception word_k...
    - exception word_0
    - exception IP
    - label

    | -4 | 0 | 0 | 0 | k | MR0 |

- Exception handler may reply with modified thread state

### L4 Protocols

- Page fault
- Thread start
- Interrupt
- Preemption
- Exception
  - Asynchronous notification

### Exception Handling

- Possible responses of exception handler:
  - **retry**: reply with unchanged state
    - possibly after removing cause
    - possibly changing other parts of state (registers)
  - **continue**: reply with IP++4 (assuming 4-byte instructions)
  - **emulation**: compute desired result,
    - reply with appropriate register value and IP++4
  - **handler**: reply with IP of local exception handler code to be executed by the thread itself
  - **ignore**: will block the thread indefinitely
  - **kill**: use `ExchangeRegisters()` (if local) or `ThreadControl()` to restart or kill thread
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
    
- no effect if receiver’s bits already set
- receiver can prevent asynchronous notification by setting a flag in its UTCB

- Two ways to receiver 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 if set bits in UTCB
  
  but remember it’s asynchronous and can change at any time!