OKL4 API Overview (OKL4 2.1)

- 9 privileged system calls
  - control resources
  - can only be executed by root task
- 7 unprivileged system calls
  - provide API to applications
  - can be invoked by anyone
- 3 communication protocols
  - for kernel-user communication
  - some form of exception IPC

OKL4 API Overview

- Privileged system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl
- Unprivileged system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

Threads

- Traditional Thread
  - Execution abstraction
  - Consist of:
    - Registers (general-purpose and status registers)
    - Stack
- OKL4 thread also has:
  - Virtual registers
  - Scheduling priority, time slice and time quantum
  - Address space
- OKL4 provides for a fixed overall number of threads
  - User threads and system threads (1 idle thread per CPU)
  - User thread created by privileged root task
  - User thread deleted / allocated to address space by holder of master capability

Virtual Registers

- Kernel-defined, user-visible thread state
- Implemented as physical machine registers or memory locations
  - Depends on architecture and ABI
- Two types:
  - Thread control registers (TCRs)
    - For sharing information between kernel and user
  - Message registers (MRs)
    - Contain the message transferred in an IPC operation
Thread Control Block (TCB)

- 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 without compromising security
    - kept in user-level TCB (UTCB)
  - Includes virtual registers
  - Must only be modified via the provided library functions!
  - No consistency guarantees otherwise
  - Many fields are only modified as side effect of some operations (IPC)

User-Level TCB

- Area of memory directly accessible to thread
- Contains thread control registers:
  - PrewptDFP, PrewptCalbackFP, ErrorCode
  - UserHandle, various flags
- Contains message registers
- More about this in IPC module

Convenience APIs:
- \( \text{I4_Word}_t \text{ I4_UserDefinedHandleDF} \) \( \text{I4_ThreadId}_t \) target;
- \( \text{void} \) \( \text{I4_UserDefinedHandleDF} \) \( \text{I4_ThreadId}_t \) target,
- \( \text{I4_Word}_t \) new_value;

UTCB Array

- Address space has a fixed region called the UTCB array
  - Location fixed at address-space creation time
    - kernel-determined on ARM7/9
  - No kernel API exists for obtaining its location — note in 2.1
  - Define user-level protocol if needed
- Each UTCB is allocated at a unique location within array
  - Determined at thread creation time
    - Kernel-determined on ARM7/9

- UTILITY slot allocation, de-allocation (except on ARM7/9)
  - …according to some policy
  - ... allows donating time to thread

Thread Capabilities — New in 2.1

- Called Thread IDs for historical reasons
- Represents local (to address space) name for a local or external thread
  - No kernel API exists to obtain address-space ID from thread cap
- Thread no is index into AS's cList
  - Defined by all thread creation
  - ... according to some policy
- Two types of thread caps:
  - IPC cap
    - Allows sending IPC to thread
    - Allows denaturing time to thread
    - Thread (master) cap
    - Allows IPC and destroy
      - 2.1: destroy is still privileged
      - In 2.2 only requires cap
        - Cap controls all thread caps

ThreadControl

- Create, destroy or modify threads
  - Privileged system call (can only be performed by root task)
- Determines thread attributes
  - Id of thread permitted to control scheduling parameters
    - This is known as the target thread's scheduler
    - Note: the scheduler thread doesn't actually perform scheduling!
  - Page fault handler ("super")
  - Exception handler
  - Access to hardware resources (e.g., FP registers)
  - Location of thread's UTCB in UTCB array (at thread creation)
    - Not on ARM7/9

Thread and Stacks

- Kernel does not allocate or manage stacks in any way
  - Only preserves IP, SP on context switch
- User level (savers) must manage
  - Stack location, allocation, size
  - Entry point address
  - Thread ID allocation, de-allocation
  - UTCB slot allocation, de-allocation (except on ARM7/9)
- Be aware of stack overflow
  - Very easy to grow stack into other data

Create, destroy or modify threads
- Privileged system call (can only be performed by root task)
- Determines thread attributes
  - Id of thread permitted to control scheduling parameters
    - This is known as the target thread's scheduler
    - Note: the scheduler thread doesn't actually perform scheduling!
  - Page fault handler ("super")
  - Exception handler
  - Access to hardware resources (e.g., FP registers)
  - Location of thread's UTCB in UTCB array (at thread creation)
    - Not on ARM7/9

Thread and Stacks
- Kernel does not allocate or manage stacks in any way
  - Only preserves IP, SP on context switch
- User level (savers) must manage
  - Stack location, allocation, size
  - Entry point address
  - Thread ID allocation, de-allocation
  - UTCB slot allocation, de-allocation (except on ARM7/9)
- Be aware of stack overflow
  - Very easy to grow stack into other data
ThreadControl

- Create, destroy and modify thread attributes
- C language API:
  ```c
  L4_Word_t L4_ThreadControl( L4_ThreadId_t target,
  L4_SpaceId_t  space,
  L4_ThreadId_t scheduler,
  L4_ThreadId_t pager,
  L4_ThreadId_t excpt_handler,
  L4_Word_t resources,
  void *utcb);
  ```

Example: Creating a Thread

```c
Create thread in address space addr_spc
L4_ThreadId_t number = …; /* Clist slot according to policy */
L4_ThreadId_t thread = L4_GlobalId(number, 1);
void *utcb = utcb_base;
if (!L4_UtcbIsKernelManaged())
utcb += L4_GetUtcbSize() * number;
else
  utcb = ~0UL;
L4_Word_t resources = 0;
L4_ThreadControl ( thread, /* new TID */
  addr_spc, /* address space to create thread in */
  scheduler, /* scheduler of new thread */
  pager, /* pager of new thread */
  exc_hdlr, /* exception handler */
  resources, /* thread resources */
  utcb); /* utcb address */
```

OKL4 API Overview

- Privileged system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl
- Unprivileged system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

ExchangeRegisters

- C language API:
  ```c
  L4_ThreadState_t L4_ExchangeRegisters( L4_ThreadId_t target,
  L4_Word_t     control,
  L4_Word_t     sp,
  L4_Word_t     ip,
  L4_Word_t     flags,
  L4_Word_t     usr_data,
  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_data,
  L4_ThreadId_t *old_pager);
  ```

- is an arbitrary user-defined value
- can be used to implement thread-local storage
- Flag allows setting processor status bits
- Can also inspect thread's GP registers
- Contents are in message registers

Example: ExchangeRegisters

```
L4_ThreadState_t L4_Stop( L4_ThreadId_t target);
void L4_Start_SpIp( L4_ThreadId_t thread,
  L4_Word_t sp,
  L4_Word_t ip);
L4_ThreadState_t L4_Start_SpIp(thread, stack, function);
```
### OKL4 API Overview

- **Privileged system calls**
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - SecurityControl
  - InterruptControl
  - ThreadSwitch
  - mutex
  - MemoryCopy
  - SpaceSwitch

- **Unprivileged system calls**
  - ExchangeRegisters
  - Schedule
  - Intr
  - ThreadSwitch
  - MemoryCopy
  - SpaceSwitch

### OKL4 IPC Operation

- Message passing in OKL4 is always synchronous (rendez-vous)
  - Message gets transferred when sender and receiver are both ready
  - The party attempting the operation first blocks until the other is ready

#### Implications:
- Implicit synchronisation
- No buffering of data in the kernel
- Data copied at most once

### IPC Overview

- Single IPC syscall incorporates a send and a receive phase
  - Both are atomic
  - Either can be omitted
  - Failure in send aborts receive

- Send operation must:
  - Specify a specific thread to send to
  - Receive operation can:
    - Specify a specific thread from which to receive ("closed receive")
    - Specify willingness to receive from any thread ("open wait")

- Each phase (send and receive):
  - Can be blocking — blocks until the partner is ready
  - Can be polling — will fail immediately if the partner is not ready

### IPC: Logical Operations

- Combine send and receive in single system call
- Result in five different logical operations
  - Send(): send message to specified thread (blocking)
  - Receive(): receive message from specified thread (blocking)
  - Call(): send message to specified thread and wait for reply from same thread
  - Typical client operation (blocking send, blocking receive)
  - Typical server operation (non-blocking send, blocking receive)

### Typical Use: Client-Server Scenario

**Client**
- Directed, blocking send
- Directed, blocking receive
- Receive immediately after send

```c
while (!done) {
  send(server, request, block);
  recv(server, reply, block);
}
```

**Server**
- Directed, non-blocking send
- Undirected, non-blocking receive
- Receive immediately after send

```c
while (1) {
  send (client, reply, !block);
  recv (&client, req, block);
}
```

### 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 message tag, defining message size (etc)
  - Rest un-typed words, not (normally) interpreted by kernel
  - Kernel protocols define semantics in some cases
**IPC Operation**

- IPC just copies data from sender’s to receiver’s MRs
  - This case is highly optimized in the kernel ("fast path")
  - For MRs backed by physical registers, it is a no-op (on same CPU)
  - Note: no page faults possible during transfer (registers don’t fault!)

**Message Tag MRs**

- Specifies message content
  - u: number of words in message (excluding MRs)
  - m: specifies memcpy operation (later)
  - r: asynchronous notification operation (later)
  - f: blocking receive
  - if unset, fail immediately if no pending message
  - b: blocking send
  - if unset, fail immediately if receiver not waiting
  - label: user-defined (e.g., opcode)

- Kernel protocols define this for some messages

**Reply Caps**

- On call-type IPC (and only there), kernel creates a reply cap
  - delivered to receiver’s l4_call argument
  - only works with wait-type receive (L4_Wait() or L4_ReplyWait())
  - receiver can use this as the cap in a send operation

- Note: Server will normally use L4_ReplyWait() rather than separate syscalls

**A Word About Protocols**

- Any communication requires protocols
  - Human communication protocols:
    - meet in certain place at a certain time (14:00 in Room Seminar-Room W)
    - use a certain medium (phone, face-to-face)
    - use a certain language (English, Swahili, Esperanto…)
    - rules about who speaks when (lecture, chaired discussion, …)
  - Similarly with programs:
    - identify communication partners (name service, “built-in”)
    - define message formats
    - define message sequences
    - define failure modes

- Clients are given rights (i.e. caps) for invoking server
  - Server should be able to:
    - know which clients it has (i.e. who has caps to it)
    - keep track of past client invocations (unless required by nature of service)
  - Server should be able to:
    - interfere with client except on client’s request
    - provide to the server by the kernel in a call-type IPC
    - can only be used once
    - can only be used by the thread (server) which received the call

- Clients are not required to:
  - keep track of past client invocations (unless required by nature of service)
  - provide to the server by the kernel in a call-type IPC
  - can only be used once
  - keep track of past client invocations (unless required by nature of service)
  - can only be used once

- Clients are not required to:
  - keep track of past client invocations (unless required by nature of service)
  - provide to the server by the kernel in a call-type IPC
  - can only be used once
  - keep track of past client invocations (unless required by nature of service)
  - can only be used once
Example: Sending 4 words

L4_MsgClear(&msg);
L4_Set_MsgLabel(&msg, 1);
L4_MsgAppendWord(&msg, word 1);
L4_MsgAppendWord(&msg, word 2);
L4_MsgAppendWord(&msg, word 3);
L4_MsgAppendWord(&msg, word 4);
L4_MsgLoad(&msg);
tag = L4_Send(tid);

Note: u, s, r are set implicitly by L4_MsgAppendWord and convenience function
Delivers MR0, ..., MR4 to thread tid
Note: Should use IDL compiler rather than doing this manually!

Example: Receiving

L4_MsgClear(&msg);
L4_MsgLabel(&msg);
tag = L4_Receive(tid);
L4_MsgStore(&msg);
label = L4_Label(tag);
assert(L4_UntypedWords(tag) == 4);
word1 = L4_MsgWord(&msg, 0);
word2 = L4_MsgWord(&msg, 1);
word3 = L4_MsgWord(&msg, 2);
word4 = L4_MsgWord(&msg, 3);

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

IPC Possible Errors

Error can be on send or receive
Error code stored in UTCB. Retrieved by L4_ErrorCode()

- Bits 1-4 indicate cause:
  - Possible cause:
    - NoPartner - issued when a non-blocking operation was requested and the partner was not ready
    - InvalidPartner - invalid cap:
      - destination doesn’t exist or don’t have rights to IPC to it
    - MessageOverflow – Message size exceeds system limit
    - IpcRejected - receiver doesn’t accept async message
    - IpcCancelled -Cancelled by another thread before transfer started
    - IPCAborted - Cancelled by another thread after transfer started

Asynchronous Notification

Remote event notification
- lightweight signaling mechanism
- Sets bit(s) in receiver’s notify flags bitmap
  - delivered without blocking sender
  - delivered immediately, directly to receiver’s UTCB, without receiver syscall

Asynchronous Notification

- Two ways to receive asynchronous notifications:
  - Asynchronously by checking notifyFlags in UTCB
    - but remember it’s asynchronous and can change at anytime
  - 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
      - Used by kernel for interrupt delivery — new in OKL4 2.1
**IPC: Obsoleted Features (from earlier L4 APIs)**

- **String item in message**
  - Used to send out-of-line data arbitrarily-sized and -aligned buffers
  - Issues with page faults during IPC, recursive kernel invocation...
  - Replaced by more restricted `MemCopy()` syscall — new in OKL4 2.1

- **Map/grant item in message**
  - Used to send page mappings through IPC
  - High kernel space overhead
  - Long-running operations that are problematic for real-time
  - Replaced by `MapControl()` syscall
  - Next OKL4 version will have non-privileged, delegatable mappings

- **Timeouts on IPC**
  - Limit blocking time
  - Practically not very useful for lack of good rules for choosing timeouts
  - Replaced by send/receive block bits (`s`, `r` respectively)
  - Use watchdog time for implementing timeouts at user level

**OKL4 API Overview**

- **Privileged system calls**
  - `ThreadControl`
  - `SpaceControl`
  - `MapControl`
  - `CapControl`
  - ` MutexControl`
  - `InterruptControl`
  - `SecurityControl`
  - `CacheControl`
  - `PlatformControl`

- **Unprivileged system calls**
  - `ExchangeRegisters`
  - `Ipc`
  - `Schedule`
  - `ThreadSwitch`
  - `Mutex`
  - `MemoryCopy`
  - `SpaceCopy`

**Protocols**

- Page fault
- Exception
- Interrupt

**OKL4 Scheduling**

- OKL4 uses 256 hard priorities (0–255):
  - Priorities are strictly observer
  - The highest-priority runnable thread will always be scheduled

- Round-robin scheduling among threads of highest priority

**OKL4 Scheduling**

- Scheduler is invoked when:
  - the current thread's time slice expires
  - Will only schedule thread of same priority (possibly same again)
  - the current thread yields
  - Will only schedule thread of same priority (possibly same again)
  - an IPC operation blocks caller or unblocks another thread
  - but see below for exceptions...

- Scheduler is not invoked when:
  - Interrupt occurs
  - May make higher-priority thread (interrupt-handler) runnable
  - Can determine at priorities of current and interrupt thread
  - The highest-priority thread can be determined without the scheduler
  - eg. send unblocks other thread to run sender or receiver based on prio
  - switch without scheduler invocation is called direct process switch

**Kernel implements schedule inheritance — new in OKL4 2.1**

- If high-prio thread is blocked on other thread (through IPC, mutex)
  - details later...

**Schedule**

- The `Schedule()` syscall does not invoke a scheduler!
  - ... not does it actually schedule any threads

- Schedule sets/reads a thread's scheduling parameters
  - The caller must be registered as the destination's scheduler
  - Set via `ThreadControl()`

- Can change
  - priority
  - time-slice length
  - processor number
  - Only relevant on multiprocessors

- Can obtain
  - priority
  - time-slice length
  - processor number
  - remaining time slice (time left to next preemption)

**Schedule Example**

Set priority:

```c
L4_word_t dummy; int status; status=L4_schedule ( tid, 0ul, 0ul, 0ul, prono, 0ul, 0ul, &dummy); printf("L4Schedule\n",status);```

©2009 Gernot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 3.7
Pre-emption Callback

- When its time slice is exhausted, a thread is preempted
  - ... re-scheduled immediately if all other runnable threads are of lower priority
  - Note: no higher priority threads can be runnable due to hard priorities
- Thread can register a preemption callback:
  - kernel saves IP in PreemptedIp register
  - when re-scheduled, kernel sets thread’s IP to registered callback address
  - kernel disables callback (until re-enabled by thread)
  - thread can fix up state and continue
- Can be used for:
  - implementing lock-free synchronization (archs w/o synchronization instructions)
  - real-time threads checking timing invariants

ThreadSwitch

- Forfeits the caller’s remaining time slice
  - Can donate remaining time slice to specified thread
    - that thread will execute to the end of the time slice on the donor’s priority
  - If no recipient specified (or recipient is not runnable):
    - normal “yield” operation
    - kernel invokes scheduler
    - call might receive a new time slice immediately
- Directed donation can be used for
  - explicit scheduling of threads
  - implementing wait-free locks (together with preemption callbacks)

ThreadSwitch Example

- Directed switch (time-slice donation):
  - `L4_ThreadSwitch(some_thread);`
- Yield (undirected switch):
  - `L4_ThreadSwitch(L4_nilthread);`

OKL4 API Overview

- Privileged system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl
- Unprivileged system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

SpaceControl

- Create and destroy address spaces
  - Target AS is designated by unique space ID
  - Allocated according to user-level policy
- Allocate clist to address space
  - For IPC, thread control rights
- Control layout of new address spaces
  - UTSB area location (not on ARM7/9)
  - Cannot change once address space is created
SpaceControl

- Create create/destroy spaces
  \[ \text{L4_World_t L4_SpaceControl}(\text{L4_SpaceId_t target, L4_World_t *old_resources}) \]

  - Can modify address-space resources (hardware-dependent)
    - MMU ASID
    - ARM PID
  - Can specify a space pager for the address space
    - OKL4.2-only feature
    - pager ID is specified in message register MR
    - supports unprivileged version of mappings:
      - space pager can map pages into the target address space
      - details later...

Deleting Address Spaces

- Deleting an address space frees up all its resources
  - Frees its Space ID
  - Removes all memory mappings
  - However SpaceControl() does not remove threads!
  - Need to be cleaned up explicitly before calling SpaceControl()

OKL4 API Overview

- Privileged system calls
  - ThreadControl
  - MapControl
  - CacheControl
  - PlatformControl

- Unprivileged system calls
  - ExchangeRegisters
  - Ip
  - Schedule
  - ThreadSwitch
  - MutexSwitch
  - MemoryCopy
  - SpaceControl

  → Protocols
  - Page fault
  - Exception
  - Interrupt

Address Spaces

- Address spaces are created empty
  - ExchangeRegisters
  - Need to be explicitly populated with page mappings
    - Kernel does not map pages automatically (except UTCB)
    - Normally address space populated by pager on demand
    - Thread runs, faults on unmapped pages, pager creates mappings
    - OS server(s) can populate pro-actively
    - E.g. Linux (default root task) pre-maps contents of executable and initialized data

MapControl

- Creates (maps) or destroys (un-maps) page mappings
  - Normally a privileged system call
    - Privilege can be given to address space to map a specific region to others
    - privileged provided via SpaceControl() system call

  \[ \text{L4_World_t L4_MapControl}(\text{L4_SpaceId_t dest, L4_World_t control}) \]

  dest: denotes target address space
  control: determines operation of syscall

  - [r]: read operation — returns (pre-syscall) mapping info
  - [m]: modify operation — changes existing mapping
  - [n]: number of \text{map items} used to describe mappings
  - map items are contained in message registers \text{MR}_r ... \text{MR}_n

Specifying Mappings: Fpages

- A \text{map or fpage} is used to specify mapping objects
  - Generalization of hardware pages
  - Similar properties:
    - size is power-of-two multiple of base hardware page size
    - base is aligned to size
    - page of size=2^s is specified as
      \[ \text{fpage}(\text{size}) \]
  - On ARM, s ≥ 12

  → Special fpages
    - size
    - alignment

  Note: Fpages will be removed in next release
Map Item

- Specifies a mapping to be created in destination AS

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpage</td>
<td>Specifies where mapping is to occur in destination AS</td>
</tr>
<tr>
<td>physadr</td>
<td>Base of physical frame(s) to be mapped</td>
</tr>
<tr>
<td>attr</td>
<td>Memory attributes (e.g. cached/uncached)</td>
</tr>
<tr>
<td>rwx</td>
<td>Permissions (e.g. read/write/execute)</td>
</tr>
</tbody>
</table>

- Note: shifted 4 bits to support 64MB of physical AS
- If specify unsupported permission, kernel widens to smallest superset (x → r)

Superpages

- Many processors support several page sizes
  - OKL4 usually supports all sizes supported by hardware
  - Actually supported sizes can be obtained from libL4
- MapControl will automatically choose largest supported size for a mapping
  - E.g. mapping a 4MB region on ARM
- Kernel will use four 1MB super-pages
- Note: next OKL4 release will require caller to specify page size
- Example of removing policy from kernel

Task

- OKL4 API does not define a concept of a "task"
- We use it informally to mean:
  - An address space, having:
    - Space id
    - UTCB area
    - Clist
    - Other resources such as space pager, ASID, ...
  - A set of threads inside that address space, each having:
    - Local thread id (clist index)
    - UTCB location
    - IP, SP
    - Pager
    - Scheduler
    - Exception handler
  - Code, data, stack(s) mapped into that address space

Steps in Creating a Task

- Create a new address space (AS)
  - SpaceControl() system call
  - Determines space ID and address-space layout according to policy
  - Associates a clist with the AS
  - Reserves virtual-memory region for UTCB array
- Map memory into AS
  - MapControl() system call
  - Maps text, data, stack(s)
- Can also be done lazily (by pager in response to page faults)
- Create threads
  - ThreadControl() system call
  - As discussed earlier
- Start first thread
  - ExchangeRegisters() system call
  - Gives thread IP, SP to make it runnable
  - First thread may start any further threads itself

Creating a Task...

Define UTCB area location in new address space

```c
L4_SpaceControl(task, /* new TID */
L4_SpaceCtrl_new, /* control */
clist, // capability list
utcb_fpage /* location of UTCB array */
0, /* no resources */
&old_resources);
```

Mapping Memory to Task

```c
L4_Fpage_t
fpage = L4_Fpage(vaddr, size);
fpage = L4_FpageAddRights(fpage, L4_Readable);
L4_PhysDesc_t
physdesc = L4_PhysDesc(paddr, L4_DefaultMemory);
L4_MapPage(space, fpage, physdesc);
```
Adding Threads to Task

- Use ThreadControl() to add new threads to AS
  ```c
  threadno = j; /* according to policy */
  thread = L4_GlobalHThreadno, i);
  utcb = j; /* according to policy, ignore on ARM#11 */
  L4_ThreadingControl(tid, task, mnu, pager, mnu, ram, utcb);
  ```
- Note: Maximum number of threads defined at address-space creation time
  - via the size of the UTCB area

Practical Considerations

- Sequence for creating tasks may seem 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 ISL compiler (Magpie)
- Note: Some older L4 APIs do not support space IDs

OKL4 API Overview

- Privileged system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl
- Unprivileged system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

Capabilities and Clists

- Caps specify communication rights
  - in future versions of OKL4 this will be extended to all system resources
- Threads can always receive messages from anywhere
- Threads can only send messages to threads whose IPC caps they hold

Capabilities

- Each address space has one clist
  - Not all caps of that address space
    - caps provide local names for kernel objects
    - Clist is a kernel object
    - not directly accessible to user code
    - Clist created/deleted by CapControl()”
- only empty clists can be destroyed
- Thread caps created by ThreadControl()”
  - deposits thread cap in slot in caller’s clist
  - identified by thread-no field in thread cap
- IPC caps created by CapControl()”
  - creates an IPC cap from a thread cap
  - deposits the IPC cap in specified clist
- Thread/IPC caps are location-transparent
  - cannot infer thread’s address space from cap
Managing Clists

```c
ok = L4_CreateClist( clist, /* id of new clist */
        n_entries); /* clist size */
ok = L4_CreateIpcCap(tid, /* thread cap */
        tid_clist, /* clist of tid */
        dest, /* new IPC cap */
        dest_clist); /* clist for dest cap */
```

Those convenience functions use CapControl() as clists determine access rights. In general an address space doesn’t have access to its own clist.
- This is presently ensured by CapControl() being a privileged system call.
- In future versions, clists will be subject to access control instead.

Privileged system calls
- ThreadControl
- SpaceControl
- MapControl
- CapControl
- MutexControl
- InterruptControl
- SecurityControl
- CacheControl
- PlatformControl

Unprivileged system calls
- ExchangeRegisters
- Ipc
- Schedule
- ThreadSwitch
- Mutex
- MemoryCopy
- SpaceSwitch

Protocols
- Page fault
- Exception
- Interrupt

Mutex

Kernel-supported mutual-exclusion mechanism

- Two kinds of mutexes supported
  - Kernel mutex: lock/unlock are system calls
    - legacy, do not use
    - System call overhead too high for uncontended locks
  - Hybrid mutex: combination of library and syscall — new in OKL 2.1
    - User-level implementation of lock/unlock (in shared memory) if uncontended
      - thread waiting on lock is put to sleep, with schedule inheritance, fairness
  - Three operations:
    - lock: acquire blocking
    - trylock: acquire non-blocking
    - unlock: release

Mutex Use

Hybrid mutex variable contains user-level state + reference to kernel mutex
- If lock operation finds mutex locked, performs Mutex() syscall to sleep
- If unlock operation finds mutex locked contended, performs Mutex call to unlock

Note: hybrid mutexes are only simulated in OKL 2.1 (always do syscall)

User-to-User Memory Copy

New in OKL 2.1

- Supports bulk data transfer without limitations of alternatives:
  - Copy server
    - Requires trusted third party
    - Higher synchronisation overhead
  - Shared memory buffer
    - page-alignment requirement
    - Space overhead of at least one page per pair of address spaces
  - Replacement for ‘long IPC’ feature of L4 V2, V4
    - Avoids drawbacks of long IPC:
      - Page faults during syscall, recursive syscalls
      - Tricky corner cases in semantics, high implementation complexity
MemoryCopy Operation
- Semi-synchronous copy between address spaces
  - similar in style to asynchronous notification:
    - one thread sets up
    - other thread invokes transfer
  - synchronous to invoker, asynchronous to initiator
- Sandwiched between IPCs
  - serve for
    - synchronization
  - don't touch buffer
  - sender between IPCs
  - receiver after final IPC
- Copy direction
  - independent of IPC direction

Initiator: Tag: L4_Send(transferer);
Transferer: MemoryCopy descriptor is in message registers, after any untyped words
  - specifies address and size of buffer
  - specifies permitted copy direction (from, to or both)

MemoryCopy Use
- Initiator:
  L4_MemCopyInitiator( msg);
  /∗ Set m bit in tag word ∗/
  L4_MemCopyWord( msg, abuf ); /∗ buffer address ∗/
  L4_MemCopyWord( msg, n); /∗ buffer size (byte) ∗/
  L4_MemCopyWord( msg, L4_MemCopyBoth); /∗ send/recv ∗/
  tag = L4_Send( transferer);
- MemoryCopy descriptor is in message registers, after any untyped words
  - specifies address and size of buffer
  - specifies permitted copy direction (from, to or both)
- Transferer:
  L4_MemCopyInitiator( src buf, n, res, L4_MemCopyFrom);
  L4_MemCopyInitiator( snd buf, n, end, L4_MemCopyTo);

InterruptControl — New in OKL4 2.1
- Manages a thread's association with interrupt handlers
  - previously part of ThreadControl() functionality
  - This is a privileged system call (despite no "Control") in its name
  - unlike earlier L4 versions, but more in line with hardware behaviour
  - different from earlier L4 versions, but more in line with hardware behaviour
- Allocate interrupt delivery by asynchronous notification
  - allows handler to decide whether to block or poll
- 5 different operations
  - register: associates IRQ(s) to thread
  - unregister: removes association of thread with IRQ(s)
  - acknowledge: acknowledge and clear interrupt at hardware
  - acknowledgeOnBehalf: ask a different thread in same address space
  - acknowledgeWait: ask and wait for next interrupt notification

SpaceSwitch — New in OKL4 2.1
- Migrates a thread between address spaces
  - previously part of ThreadControl() functionality
  - in its name
  - unlike earlier L4 versions, but more in line with hardware behaviour
  - different from earlier L4 versions, but more in line with hardware behaviour
- Allocate interrupt delivery by asynchronous notification
  - allows handler to decide whether to block or poll
- 5 different operations
  - register: associates IRQ(s) to thread
  - unregister: removes association of thread with IRQ(s)
  - acknowledge: acknowledge and clear interrupt at hardware
  - acknowledgeOnBehalf: ask a different thread in same address space
  - acknowledgeWait: ask and wait for next interrupt notification

MemoryCopy Operation
- Semi-synchronous copy between address spaces
  - similar in style to asynchronous notification:
    - one thread sets up
    - other thread invokes transfer
  - synchronous to invoker, asynchronous to initiator
- Sandwiched between IPCs
  - serve for
    - setup
  - synchronization
  - don't touch buffer
  - sender between IPCs
  - receiver after final IPC
- Copy direction
  - independent of IPC direction

OKL4 API Overview
- Privileged system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl
- Protocols
  - Page fault
  - Exception
  - Interrupt

OKL4 API Overview
- Privileged system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl
- Protocols
  - Page fault
  - Exception
  - Interrupt
InterruptControl Use

L4_RegisterInterrupt(
    L4_Thread_t thread, /* handler thread, local to address space */
    L4_Word_t n_bit, /* number of bit used for interrupt notification */
    L4_Word_t mrs, /* number of highest map reg for parameters */
    L4_Word_t request); /* additional parameter */

L4_AcknowledgeInterrupt(L4_Word_t mrs, L4_Word_t request);

Details of parameters are platform-specific
- defined by board support package
- typically deal with one IRQ at a time
- platform-defined parameter format
  - parameters in MR0, MR1
- typically word used for interrupt notification
- additional parameter not needed by most platforms

OKL4 API Overview

→ Privileged system calls
  • ThreadControl
  • SpaceControl
  • MapControl
  • CapControl
  • MutexControl
  • InterruptControl
  • SecurityControl
  • CacheControl
  • PlatformControl
  → Protocols
    • Page fault
    • Exception
    • Interrupt

→ Unprivileged system calls
  • ExchangeRegisters
  • Ip
  • Schedule
  • ThreadSwitch
  • Mutex
  • MemoryCopy
  • SpaceSwitch

SecurityControl — New in OKL4 2.1

Used to delegate certain privileges to an address space
- enables unprivileged address space to perform limited privileged operations
- logically derives a capability and grants this to target address space
- will become redundant once all resources are explicitly capability-controlled

Deals with 5 different privileges
- platform: grants right to invoke PlatformControl()
- space-switch: grants right to switch threads to another address space
  - specifies the allowed target address space
- interrupt: assigns specified interrupts to an address space
  - enabling it to associate those using InterruptControl()
- map: grants rights to an address space to map memory to a different space
  - specifies the physical memory region the address space can map
    - enables the address space to invoke MapControl() for that region
      - but only map to an address space whose space pager is the caller
  - domain: grants rights to set up shared regions using a common domain ID
    - supports high-performance sharing on ARM arch (architecture-specific)

CacheControl

Controls state of CPU caches
- may operate on complete cache
- may operate on all cache lines holding data of a certain memory region
- may operate on instruction or data cache
- may operate on all or specified levels of cache

6 different operations
- flush: clean any modified lines corresponding to region, then invalidate
- lock: lock lines corresponding to region
- unlock: unlock lines corresponding to region
- flushI: clean and invalidate complete instruction cache
- flushD: clean and invalidate complete data cache
- flushAll: clean and invalidate all caches

Example: Driver flushes I/O buffer prior to DMA
  • L4_CacheFlushRange(space, start_addr, end_addr);
PlatformControl

- Platform-specific system call
- Used for platform-specific functionality
- Presently used only for power management
- Used to set core voltage and frequency
- Right to use may be granted to unprivileged address space using SecurityControl()?

Possible use:

```
l4_PlatformControl(0, bus_freq, cpu_freq, voltage);
```

OKL4 API Overview

- Privileged system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl

- Unprivileged system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceCopy

Page Fault Handling

- Address-spaces are populated in response to page faults
- Page faults are converted into IPC messages:
  1. Thread triggers page fault
  2. Kernel exception handler generates IPC from fault to pager
  3. Pager establishes mapping
     - Calls MapControl()
     - If not privileged to do this, has to ask root task
  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 messages

```
<table>
<thead>
<tr>
<th>Fault IP</th>
<th>MR_0</th>
<th>MR_1</th>
<th>MR_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2002</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- E.g. page fault at address 0x2002: Kernel sends

```
<table>
<thead>
<tr>
<th>Fault IP</th>
<th>MR_0</th>
<th>MR_1</th>
<th>MR_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2002</td>
<td>0x12</td>
<td>0x00</td>
<td>0x00</td>
</tr>
</tbody>
</table>
```

- Application could manufacture same message if it had a cap to the pager
  - Provided is has a cap to IPC to the pager (which it generally does not have)
  - Pager could not tell the difference
  - Could mess up OS bookkeeping (has page been mapped?)
  - Better not to give apps a cap to their pager (possible in OKL4 2.1)

Pager Action

- E.g. pager handles write page fault at 0x0200
  - Map item to map 4KB page at PA 0x0000

```
<table>
<thead>
<tr>
<th>Page Faulting Address</th>
<th>Page Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0           12       0</td>
<td></td>
</tr>
<tr>
<td>0x0000         0x300    0</td>
<td></td>
</tr>
</tbody>
</table>
```

- Note: phys addr must be aligned to page size
- Next, pager replies to page-fault message
- Content of message completely ignored
- Serves for synchronization: informing kernel that fault can be restarted
- If pager did not establish mapping, client will trigger same fault again
OKL4 Protocols

- Page fault
- Exception
- Interrupt

Exception Protocols

- Other exceptions (invalid instructions, division by zero…) result in a kernel-generated IPC to thread’s exception handler
- Exception IPC
  - Kernel sends (partial) thread state
    - Exception word
    - Exception IP
- Other exceptions (invalid instructions, division by zero…) result in a kernel-generated IPC to thread’s exception handler
- Exception IPC
  - Kernel sends (partial) thread state
    - Exception word
    - Exception IP
- Label:
  - 4: Standard exception, architecture independent
  - 5: Architecture-specific exception

Exception Handling

- Possible responses of exception handler:
  - retry: reply with unchanged state
  - continue: reply with IP+=4 (assuming 4-byte instructions)
  - emulation: compute desired result
  - 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

Exception State

- Thread state sent to exception handler depends on exception
  - General exception: kernel sends:
    - IP, SP, CPSR, exception number
    - Error code (exception specific)
  - VFP exception (ARM): kernel sends:
    - IP, SP, CPSR, exception number
    - Error code (Exception specific)
    - Faulting FP instruction, next instruction, FP SCR (status word)
  - Syscall exception: kernel sends:
    - IP, SP, LR, CPSR, R0, …, R7
    - On ARM: syscall instruction

Exception Reply

- Thread remains blocked until exception handler replies
  - Logically performs call() IPC
  - Reply has same format
    - Kernel uses to overwrite thread state
    - To leave unchanged, send same message back
    - Obviously kernel will not let you modify privileged state (eg in status register)

OKL4 Protocols
What is in a device?

- **Registers**
  - The interface to the device
  - Provides the mechanism for modifying device state
  - Provides the mechanism for querying the device state

- **Interrupts**
  - Mechanism for device to notify CPU of events

- **Direct Memory Access**
  - Allows device to access memory efficiently
  - Compare with programmed I/O (PIO)

Device Registers

- **Memory mapped**
  - Mapped into the hardware address space
  - Can be accessed using normal memory operations
  - Should map with caching disabled!
  - I/O Ports (x86 only)
    - Separate I/O address space
    - Uses special instructions (INB, OUTB)
  - Register size
    - Usually word size
    - Can be smaller: e.g. 8-bit or 16-bit
    - Important to use the correct types to avoid incorrect operation

Interrupts

- **Used to signal event**
  - Network packet arrived
  - Disk operation completed

- **Device is usually connected to an interrupt controller**
  - Interrupt controller multiplexes many interrupt sources onto CPU interrupt line
  - Two different options for L4:
    - Export just the CPU interrupt, demultiplex interrupt sources at user level
    - Kernel demultiplex interrupt source, export each individual interrupt source
    - Both have pros and cons: decision is platform specific

- **Board support package may update interrupt descriptor in handler's UTCB**
  - Can be used to pass additional information to driver

Interrupt Handlers

- **Typical setup:** Interrupt handler is “bottom-half” device driver

  1. Interrupt is triggered, hardware disables interrupt and invokes kernel
  2. Kernel masks interrupt, determines interrupt number and notifies handler
  3. Handler identifies interrupt cause by inspecting notify mask, possibly inspecting interrupt descriptor in handler's UTCB (if set by BSP)
  4. Handler acknowledges interrupt via InterruptControl()
  5. Handler queues request to top-half driver
  6. Handler sends notification to top-half, waits for next interrupt (reply-and-wait (PC))

Direct Memory Access

- **DMA is important for performance of bulk-I/O devices**
  - Actual I/O happens bypasses the CPU
  - However still impacts performance because I/O consumes memory bus cycles
  - DMA is not necessarily cache coherent
    - DMA engine works directly on the physical memory, bypasses the CPU cache
    - Potential for incorrect data to be read by the device
  - IA32: The exception, DMA is cache coherent
  - Must explicitly flush cache before performing DMA (except on IA32!)
    - Use CacheControl()
    - Not needed on x86

  Device driver must translate a virtual addresses to physical addresses
  - Can use L4_MapControl to query pagetable
  - Usually root server keeps track of these mappings to avoid a kernel call
**OKL4 Protocols**

- Page fault
- Exception
- Interrupt

**Operating System**

- User Task
- User-defined
- Page
- Exception handler
- Special handler
- Pager exception handler

**OKL4 API Overview**

- Privileged system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl

- Unprivileged system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch

**Protocols**

- Page fault
- Exception
- Interrupt

**Final Remarks**

Beer o'clock!