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:
  - PREemptP, PREemptCALLbackP, ErrorCode
  - Userhandle, various flags
- Contains message registers
- More about this in IPC module
- Convenience APIs:

```
14_Word_t 14_UserDefinedHandle(14_ThreadId_t target);
void 14_userDefHandle(14_ThreadId_t target, 14_Word_t new_value);
```

Thread Capabilities — New in 2.1

- Called Thread ids for historical reasons
- Represent 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 thread creation
  - ...according to some policy
- Two types of thread caps:
  - IPC cap
    - allows sending IPC to thread
    - allows domaining time to thread
    - thread (master) cap
  - allows IPC and destroy
  - 2.1: destroy is still privileged
  - In 2.2 will only require cap: cap controls all thread caps

Thread Control

- Create, destroy or modify threads
  - Privileged system call (can only be performed by root tasks)
- 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 ("page")
  - exception handler
  - access to hardware resources (eg FP registers)
  - location of thread's UTCB in UTCB array (at thread creation)
  - not on ARM7/9

- From CH4 2.2 on:
  - Create only requires resources
  - Changing attributes only requires master cap

Threads and Stacks

- Kernel does not allocate or manage stacks in any way
- only preserves IP, SP on context switch
- User level (sensors) 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 manage threads.
- 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_Word_t number = …; /* Clist slot according to policy */
L4_ThreadId_t thread = L4_GlobalId(number, 1);
L4_ThreadId_t scheduler, /* scheduler of new thread */
L4_ThreadId_t pager, /* pager of new thread */
L4_ThreadId_t excpt_handler; /* exception handler */
L4_Word_t resources; /* thread resources */
L4_ThreadControl ( thread, addr_spc, /* new TID */
  scheduler, /* address space to create thread in */
  pager, /* scheduler of new thread */
  excpt_handler, /* pager of new thread */
  resources, /* thread resources */
  utcb); /* utcb address */
```
OKL4 API Overview

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

- Unprivileged system calls
  - ExchangeRegisters
  - Schedule
  - Message
  - MemoryCopy
  - SpaceSwitch

- Protocols
  - Page fault
  - Exception
  - Interrupt

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 to receive from
    - allow willing to receive from any thread

- 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)
    - Wait(): receive message from any thread (blocking)
    - Call(): send message to specified thread and wait for reply from same thread
      - Typical client operation (blocking send, blocking receive)
    - Reply and Wait(): send message to specified thread, wait for message from any
      - Typical server operation (non-blocking send, blocking receive)

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!)

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
    - …

Message Tag MR0

- Specifies message content
  - u: number of words in message (excluding MR0)
  - m: specifies memcpy operation (later)
  - n: specifies asynchronous notification operation (later)
  - r: 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)
  - kernel protocols define this for some messages

Example: Sending 4 words

```c
L4Msg_t msg;
L4MsgTag_t tag;
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

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

IPC Result MR0

- Returned from any IPC operation
  - E: error occurred, check ErrorCode in UTCB
- Some fields are only useful on a receive operation, and define the received message
  - label: label of message sent (copy of label specified by sender)
  - u: number of untyped words received
  - X: message came from another CPU

Example: Receiving

```c
L4Msg_t msg;
L4MsgTag_t tag;
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()
- Lower bit indicates send or receive phase. (Can’t be both!)
- Bits 1-4 indicate cause:
  - NoPartner - issued when a non-blocking operation was requested and the partner was not ready
  - InvalidInit - invalid cap

### Asynchronous Notification

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

### IPC: Obsoleted Features (from earlier L4 APIs)

- String item in message
  - Used to send out-of-line data arbitrarily-sized and aligned buffers
  - Issued 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
- Timouts on IPC
  - Limit blocking time
  - Practically not very useful for lack of good rules for choosing timeouts
  - Replaced by send/receive block bits (r, respectively)
  - Use watchdog time for implementing timeouts at user level

### OKL4 API Overview

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

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

- Protocols
  - Page fault
  - Exception
  - Interrupt
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 looking at priorities of current and interrupt thread
- The highest-priority thread can be determined without the scheduler
  - eg. send unblocks other thread or run sender or receiver based on prio

Kernel implements schedule inheritance — new in OKL4 2.1
- if high-priority thread is blocked on other thread (through IPC, mutex)
  - details later...

Schedule

The Schedule() syscall does not invoke a scheduler!
- nor 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)

OKL4 API Overview

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

Protocols
- Page fault
- Exception
- Interrupt

Schedule Example

```c
L4_word_t dumy;
int status;
status=L4_schedule( tid,
  0ul, 0ul, 0ul,
prio, &dumy, &dumy);
printf("status=%d\n",status);
```

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)
- 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):
  ```c
  L4_ThreadSwitch(some_thread);
  ```
- Yield (undirected switch):
  ```c
  L4_ThreadSwitch(L4_nilthread);
  ```

SpaceControl

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

SpaceControl

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

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
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - SecurityControl
  - CacheControl
  - PlatformControl

- Protocols
  - Page fault
  - Exception
  - Interrupt
Address Spaces

- Address spaces are created empty
  - Except UTCB
- 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 mapping
- OS server(s) can populate proactively
  - E.g., /sbin/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
    - Bad privilege can be given to address space to map a specific region to others
    - Privileged provided via SpaceControl() system call

MapItem

- Specifies a mapping to be created in destination AS

Superpages

- Many processors support several page sizes
  - OKL4 usually supports all sizes supported by hardware
  - Actually supported sizes can be obtained from SbH
- MapControl will automatically choose largest supported size for a mapping
  - 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
    - other resources such as space pager, ASID, ...
  - a set of threads inside that address space, each having
    - local thread id (thread index)
    - UTCB location
    - FP, SP
    - scheduler
    - exception handler
  - code, data, stack(s) mapped into that address space
Steps in Creating a Task

1. 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

2. Map memory into AS
   - MapControl() system call
   - Maps text, data, stack(s)
   - Can also be done lazily (by pager in response to page faults)

3. Create threads
   - ThreadControl() system call
   - As discussed earlier

4. 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
L4_SpaceControl( task, /* new TID */
L4_SpaceCtrl_new, /* control */
clist, /* capability list */
ucb_fpage /* location of UTCB array */
0, /* no resources */
old_resources);

Mapping Memory to Task

L4_Fpage_t fpage = L4_Fpage(vaddr, size);
fpage = L4_FpageAddRights(fpage, L4_Readable);
L4_PhysDesc_t physdesc = L4_PhysDesc(paddr, L4_DefaultMemory);
L4_MapFpage(space, fpage, physdesc);

Adding Threads to Task

Use ThreadControl() to add new threads to AS
threadno = ...; /* according to policy */
thread = L4_GlobalId(threadno, 1);
utcb = ...; /* according to policy, ignore on ARM9/11 */
L4_ThreadControl(tid, task, me, pager, me, res, utc b);

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 CIL 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

→ Caps specify communication rights
  - supports information-flow control
  - in future versions of OKL4 this will be extended to all system resources
  - Threads can only send messages to threads whose IPC caps they hold

Managing Clists

```c
ok = L4_CreateClist( clist, n_entries); /* id of new clist */
ok = L4_CreateIpcClist( tid_clist, tid, dest_clist); /* id of newIPC 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

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

→ Each address space has one clist
  - holds all cists of that address space
  - caps provide local names for kernel objects
→ Clist is a kernel object
  - not directly accessible to user code
→ Clist is created/deleted by CapControl()
  - only empty cists can be destroyed
→ Thread.caps created by ThreadControl()
  - deposits thread cap in clist in caller's clist
→ 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

Mutex

→ Kernel-supported mutual-exclusion mechanism
→ Two kind of mutexes supported
  - kernel mutex: lock/unlock are system calls
    - legacy, do not use
  - hybrid mutex: combination of library and syscall — new in OKL 2.7
    - user-level implementation of lock/unlock (in shared memory) if not ccontended
    - syscall if ccontended
→ 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

MutexControl() convenience functions:

- ok = okl4_mutex_init( mutex); /* alloc kernel mutex and initialise */
- ok = okl4_mutex_free( mutex); /* free kernel mutex etc */

Mutex operation:

- ok = okl4_mutex_lock( mutex); okl4_mutex_unlock( mutex);

- 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 OKL4 2.1 (always do syscall)

User-to-User Memory Copy

New in OKL4 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, receive syscalls
    - tricky corner cases in semantics, high implementation complexity

- serves for:
  - asynchronous copy to IPA, IPCs
  - receiver after final IPC
  - synchronous to invoker, asynchronous to initiator

OKL4 API Overview

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

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

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

- independent of IPC direction

MemoryCopy Use

- Initializer:
  - L4_RegUser(int) : ansj;
  - L4_PropMemoryCopy(ansj); /* set m bit in tag word */
  - L4_RegSendWord( ansj, kbuf); /* buffer address */
  - L4_RegRecvWord(ansj, n); /* buffer size (byte) */
  - L4_MemCopyWord(ansj, L4_MemCopyToBoth<<30);/ * ansj/recv */
  - 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_MemCopy(Initiator, src_buf, n_src, L4_MemCopyFrom);
  - L4_MemCopy(Initiator, src_buf, n_src, L4_MemCopyTo);

Protocols

- Page fault
- Exception
- Interrupt
SpaceSwitch — New in OKL4 2.1

- Migrates a thread between address spaces
- previously part of ThreadControl() functionality
- This is a privileged system call (despite no "Control" in its name
  but limited switch privilege is delegatable via SecurityControl()}

L4_SpaceSwitch( L4_ThreadId_t tid, /* target thread */
L4_SpaceId_t space, /* where to */
void *utcb);/* new slot */

InterruptControl — New in OKL4 2.1

- Manages interrupts
- association/disassociation of handler threads with IRQs
- interrupt acknowledgement
- right to use must be delegated to address space via SecurityControl()}
- mostly replaces interrupt IPC protocol of earlier L4 versions
- Actual interrupt delivery is by asynchronous notification
  different from earlier L4 versions, but more in line with hardware behaviour
  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 by a different thread in same address space
  - acknowledgeWait: ask and wait for next interrupt notification

InterruptControl Use

L4_Register_Interrupt( L4_ThreadId_t thread, /* handler thread, local to address space */
L4_Word_t mrs, /* number of bit used for interrupt notification */
L4_Word_t n_bit, /* number of highest msg reg for parameters */
L4_Word_t request); /* additional parameter */
L4_AcknowledgeInterrupt( L4_Word_t mrs, L4_Word_t request);

OKL4 API Overview

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

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 targeted 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
    - deals with system calls associated with interrupts
    - uses limited switch privilege
  - 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 right to set up shared regions using a common domain ID
    - supports high-performance sharing on ARM9 (architecture-specific)
Privileged system calls
• ThreadControl
• SpaceControl
• MapControl
• CapControl
• MutexControl
• InterruptControl
• SecurityControl
• CacheControl
• PlatformControl

Unprivileged system calls
• ExchangeRegisters
• lp
• Schedule
• ThreadSwitch
• Mutex
• MemoryCopy
• SpaceSwitch

Protocols
• Page fault
• Exception
• Interrupt

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_CacheFlushDRange(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 Protocols
- Page fault
- Exception
- Interrupt
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 message

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault IP</td>
<td>MR3</td>
</tr>
<tr>
<td>Fault address</td>
<td>MR1</td>
</tr>
<tr>
<td>Error code</td>
<td>S[15..8] S[14..7]</td>
</tr>
</tbody>
</table>

  - E.g. page fault at address 0x2002: Kernel sends
  - 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 0x2002
  - Map item to map 4KB page at PA 0xc000
  - Note: phys addr must be aligned to page size

  - Next, pager replies to page-fault message
    - Content of message completely ignored
    - Serves for synchronisation: informing kernel that faulting thread can be restarted

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception word</td>
<td>MR6</td>
</tr>
<tr>
<td>Exception word</td>
<td>MR5</td>
</tr>
<tr>
<td>Exception code</td>
<td>MR4</td>
</tr>
<tr>
<td>Label</td>
<td>0</td>
</tr>
<tr>
<td>Arch</td>
<td>0</td>
</tr>
</tbody>
</table>

  - Label: -4: Standard exception, architecture independent -5: Architecture-specific exception

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 (e.g. in status register).

Exception Handling

- Possible responses of exception handler:
  - **retry**: reply with unchanged state
  - Possibly alter removing cause
  - **continue**: reply with IP+=4 (assuming 4-byte instructions)
  - Possibly changing other parts of state (registers)
  - **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

OKL4 Protocols

- **Page fault**
- **Exception**
- **Interrupt**

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, demux interrupt sources at user level
    - Kernel demux interrupt source, export each individual interrupt source
  - Both have pros and cons: decision is platform specific
    - On x86, decodes interrupt source inside the kernel
    - On ARM typically have separate interrupt for each source
Interrupts

- Modelled as asynchronous notification sent by hardware
  - received by interrupt handler thread registered for that interrupt
  - handler associated via InterruptControl()
  - handler uses InterruptControl() to acknowledge interrupt

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

Direct Memory Access

- DMA is important for performance of bulk-I/O devices
  - actual IO 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

OKL4 API Overview

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

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

- Protocols
  - Page fault
  - Exception
  - Interrupt