Open Kernel Labs™
Be open. Be safe.

OKL4 Programming
Overview of the OKL4 3.0 API

Gernot Heiser, PhD
Open Kernel Labs

Copyright Notice

Open Kernel Labs™

Be open.
Be safe.

Change overview

Be open.
Be safe.

- Removed in OKL4 3.0
  - SecurityControl system call
  - Privileged root task
  - Space pager
- Removed in OKL4 2.1
  - global thread IDs
- Removed earlier
  - string items in IPC messages
  - mapping via IPC system call

- New/changed in OKL4 3.0
  - Mappings no longer use fpages
  - memory segments new
- New/changed in OKL4 2.1
  - thread capabilities
  - clists
  - Secure HyperCells
  - scheduling inheritance
  - hybrid mutexes
  - InterruptControl, PlatformControl
  - MemoryCopy
- New/changed earlier
  - space IDs
  - kernel mutexes

OKL4 API Overview (OKL4 3.0)

Be open.
Be safe.

- → 8 resource-control system calls
  - manage system resources
  - require privilege over such resournpces
  - typically used by OS code
- → 7 other system calls
  - provide API to applications
- → 3 communication protocols
  - for kernel-user communication
  - some form of exception IPC
- → *Concept of privileged root task removed in OKL4 3.0*
  - had monopoly over resource-control system calls
  - replaced by capability-based resource management

OKL4 API Overview

- Resource-control system calls
  - *ThreadControl*
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

Threads

Be open.
Be safe.

- 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
- There are user threads and system threads
  - idle threads (1 per CPU) are the only system threads
- Threads are controlled by *capabilities* (caps) held in *capability lists* (clists)
  - user thread deleted / allocated to address space by holder of *master capability*
  - address spaces have fixed-size clists
    ⇒ limited number of threads accessible by an address space
    - local threads + threads to manipulate / IPC to

Virtual Registers

Be open.
Be safe.

- 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)

Be open.
Be safe.

- 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)

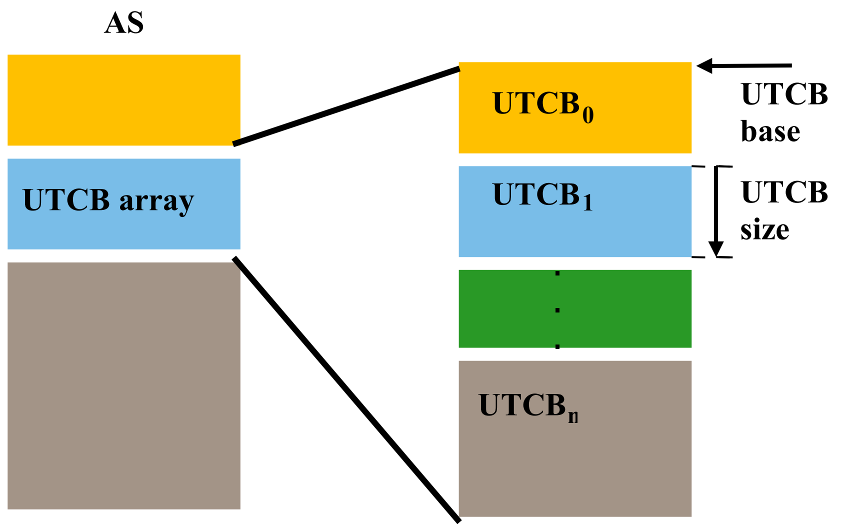
User-Level TCB

- Area of memory directly accessible to thread
- Contains thread control registers:
  - PreemptedIP, PreemptCallbackIP, ErrorCode
  - UserHandle, various flags
- Contains message registers
  - More about this in IPC module
- libl4 APIs:

```
L4_Word_t L4_UserDefinedHandleOf  ( L4_CapId_t target);
void      L4_Set_UserDefinedHandle( L4_CapId_t target,
                                    L4_Word_t  new_value);
```

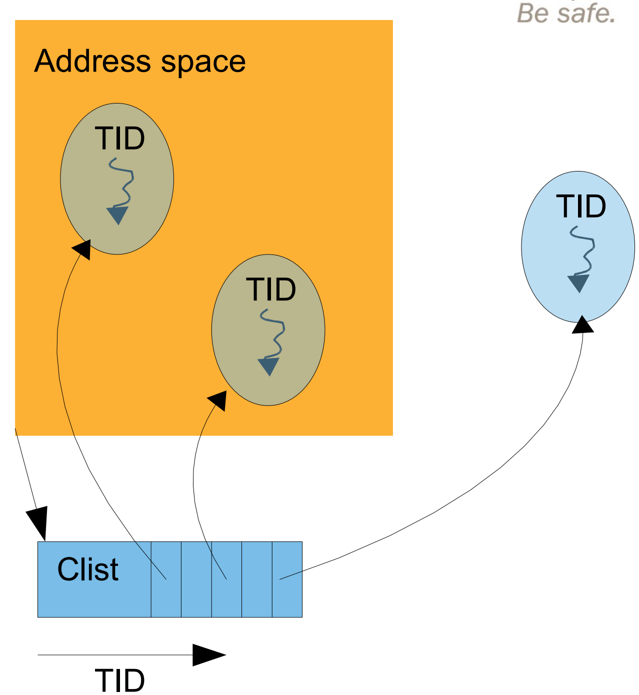
UTCB Array

- 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 — *new 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!
  - size of UTCB array limits the number of threads in the space
    - defined at space-creation time
  - size of clist limits the number of threads accessible to the space



Thread Capabilities

- *Replace former notion of Thread Ids*
- 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 at thread creation
    - …according to some policy
- Two types of thread caps:
  - IPC cap
    - allows sending IPC to thread
    - allows donating time to thread
  - thread (master) cap
    - allows IPC and destroy
    - *destroy was still privileged in OKL4 2.1*



Thread ID

Thread no (18)	1 (14)

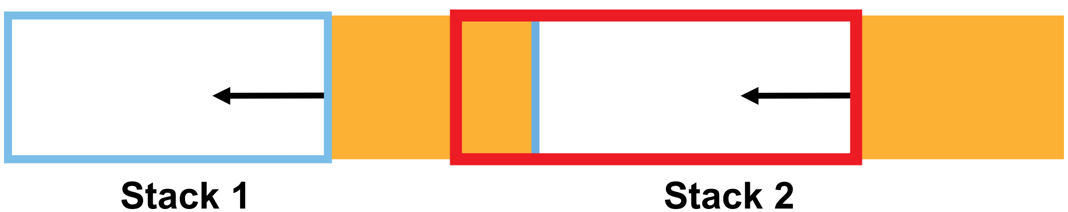
ThreadControl

Be open.
Be safe.

- Create, destroy or modify threads
  - create requires access to appropriate resources
  - destroy/modify requires master cap over thread
- 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 (*"pager"*)
  - exception handler
  - access to hardware resources (eg. FP registers)
  - location of thread's UTCB in UTCB array (at thread creation)
    - not on ARM7/9

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, de-allocation
  - UTCB slot allocation, de-allocation (except on ARM7/9)
- *Be aware of stack overflow!*
  - Very easy to grow stack into other data




```
f () {
   int buf[10000];
   . . .
}
```

ThreadControl

- Create, destroy and threads and modify thread attributes
- C language API:

```
L4_Word_t L4_ThreadControl( L4_CapId_t   target,
                            L4_SpaceId_t space,
                            L4_CapId_t   scheduler,
                            L4_CapId_t   pager,
                            L4_CapId_t   excpt_handler
                            L4_Word_t    resources,
                            void         *utcb);
```

Example: Creating a Thread

- Create thread in address space *addr_spc*

```
L4_Word_t  number = …; /* Clist slot according to policy */
L4_CapId_t thread = L4_CapId(TYPE_CAP, number);
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

Be open.
Be safe.

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - *ExchangeRegisters*
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

ExchangeRegisters

- Reads, and optionally modifies, kernel-maintained thread state
- Is used for:
  - Activating newly created threads (giving them a valid IP, SP)
  - De-activating threads
  - Multiplexing a kernel thread between several logical threads
    - Maintain a thread pool
  - Saving and restoring thread state
    - E.g: for implementing signals, checkpointing, …
- Can be executed on a thread:
  - By a thread in the same address space
  - By the thread's pager
  - By the root task

ExchangeRegisters

Open Kernel Labs

Be open.
Be safe.

- Raw C language API:

```
L4_CapId_t L4_ExchangeRegisters(L4_CapId_t   target,
                                L4_Word_t    control,
                                L4_Word_t    sp,
                                L4_Word_t    ip,
                                L4_Word_t    flags,
                                L4_Word_t    usr_data,
                                L4_CapId_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_CapId_t   *old_pager);
```

- `usr_data` is an arbitrary user-defined value
  - can be used to implement thread-local storage
- Flags allows setting processor status bits
- Can also inspect/modify the thread's GP registers
  - contents are in message registers
- *Note: Should not be used directly, use libl4 functions!*

ExchangeRegisters

- libl4 APIs:

```
L4_ThreadState_t L4_Stop(         L4_CapId_t target);
void             L4_Start_SpIp(   L4_CapId_t thread,
                                  L4_Word_t sp,
                                  L4_Word_t ip);
```

- Example: starting threads:

```
L4_Start_SpIp (thread, stack, function);
```

OKL4 API Overview

Be open.
Be safe.

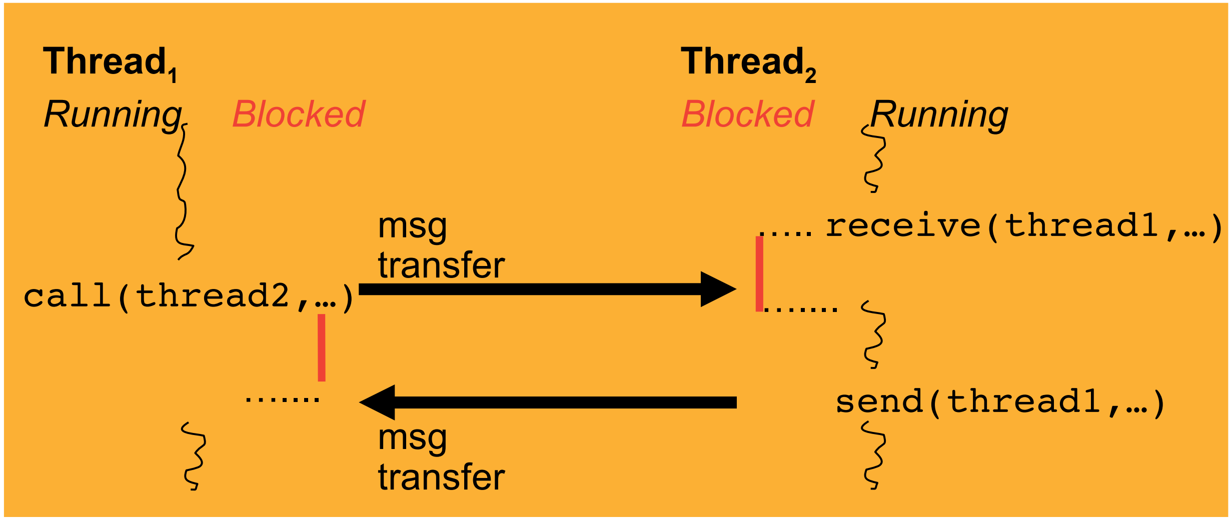
→ Resource-control system calls
- ThreadControl
- SpaceControl
- MapControl
- CapControl
- MutexControl
- InterruptControl
- CacheControl
- PlatformControl

→ Other system calls
- ExchangeRegisters
- *Ipc*
- Schedule
- ThreadSwitch
- Mutex
- 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

Open Kernel Labs™

Be open.
Be safe.

- 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

Typical Use: Client-Server Scenario

Be open.
Be safe.

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

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

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

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

IPC: Logical Operations

Be open.
Be safe.

- 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
    - delivers *reply cap* to receiver
    - 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)

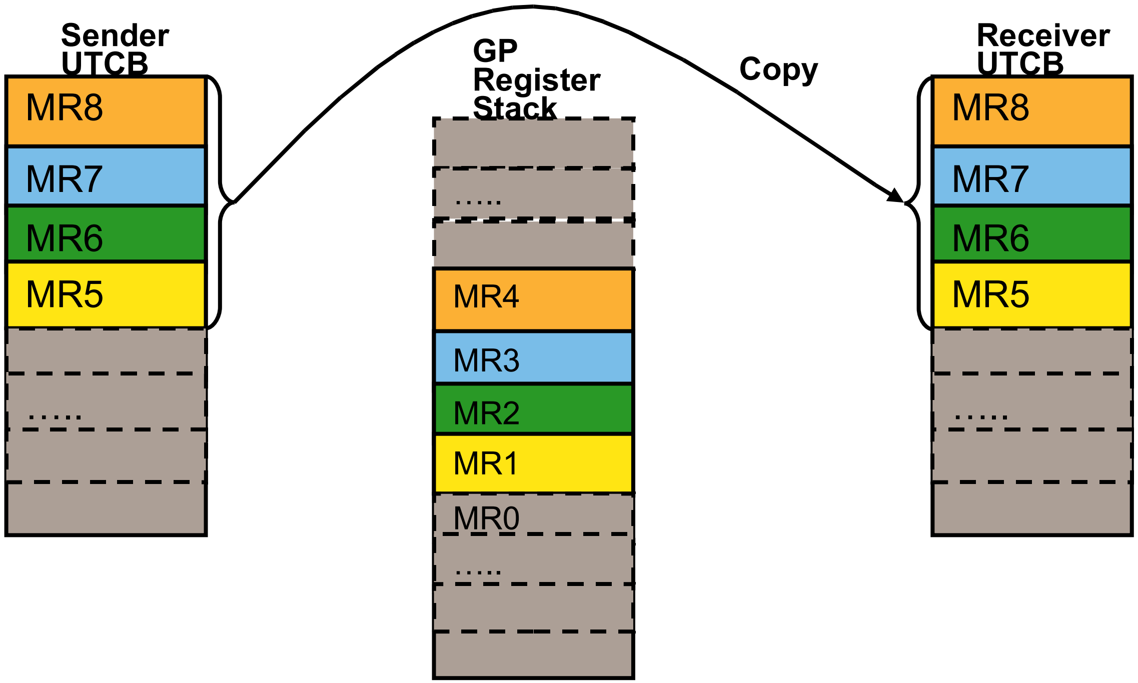
IPC Registers

Be open.
Be safe.

- 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 MR_0

label $_{(16)}$	s	r	n	m	~ $_{(6)}$	u $_{(6)}$

- Specifies message content

u: number of words in message (excluding MR_0)

m: specifies *memory copy* 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

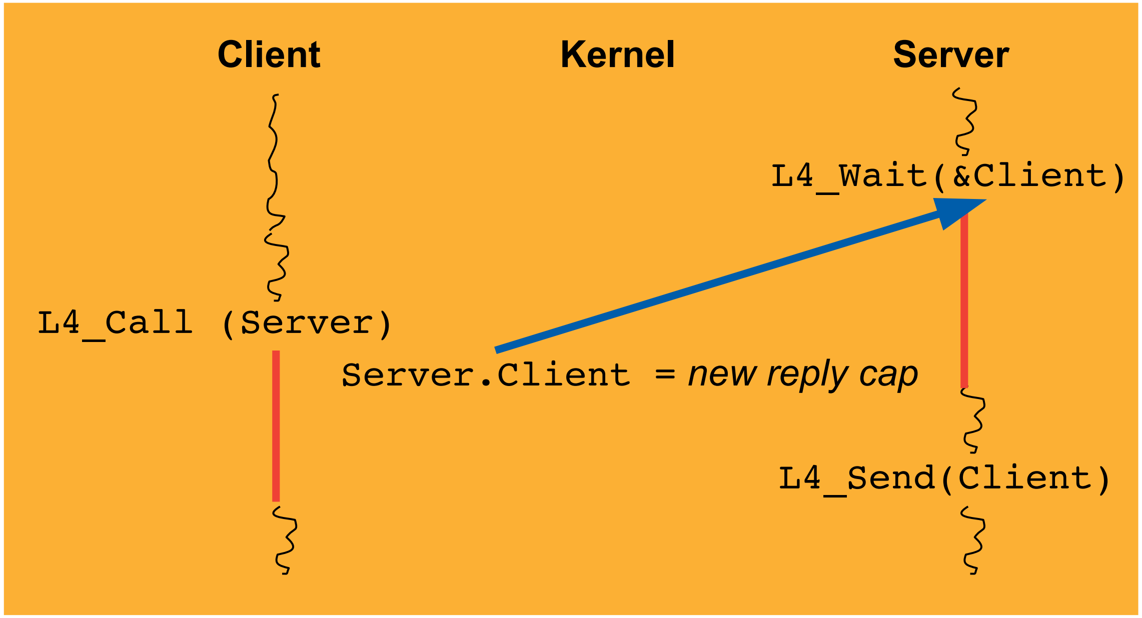
Reply Caps

Be open.
Be safe.

- Typical client-server scenario:
  - Clients are given rights (i.e. caps) for invoking server
  - Server shouldn't need 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 shouldn't be able to:
    - interfere with client except on client's request
- When client invokes server, server must be able to reply back
  - Client could provide server with cap for reply
  - However, it couldn't stop server from storing the cap for future use
- Reply caps address this issue:
  - provided 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

Reply Caps

- On call-type IPC (and only there), kernel creates a *reply cap*
  - delivered to receiver's `from` 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

Reply Caps Limitations

Be open.
Be safe.

- Reply caps have “magic” other thread caps do not have:
  - a reply cap can only be used once
  - a reply cap is manufactured by the kernel on-they-fly
    - as a side effect of an L4_Call() operation
    - not by L4_Send(), L4_ReplyWait()
    - not by any other system call
  - a reply cap is not in any clist (and therefore doesn't occupy a clist slot)
  - a reply cap can only be used in an IPC send operation
    - L4_Send()
    - L4_ReplyWait()
  - a reply cap can only be used by the thread which received it from the kernel
    - cannot be shared with other threads in the same address space
    - cannot be passed to another address space

Example: Sending 4 words

label	0	0	0	4

```
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        (receiver) ;
```

Note: *u, s, r* are set implicitly by L4_MsgAppendWord and convenience function

Delivers $MR_0, \ldots, MR_4$ to thread *tid*

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

IPC Result MR_0

label $_{(16)}$	E	X	~	~	~ $_{(6)}$	u $_{(6)}$

- 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

label	0	0	0	4

```
L4Msg_t  msg;
L4MsgTag_t  tag;

tag = L4_Receive(sender);
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

Be open.
Be safe.

- 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:
- 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
    - consequence of ExchangeRegisters

Asynchronous Notification

Be open.
Be safe.

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

Sender **Kernel** **Receiver**

```
                                      L4_Accept(L4_Acceptor(1))
tag=L4_Notify(Receiver,flags)
          Receiver.Utcb.NotifyFlags|=msg.w1
                                      flags=L4_Get_Notify_Bits()
```

- receiver must enable asynchronous notification by updating its *acceptor*
- sender-specified flags are OR-ed to receiver's flags
- accumulates: no effect if receiver's bits already set

Asynchronous Notification

Be open.
Be safe.

- Two ways to receive asynchronous notifications:
  - Asynchronously by checking NotifyFlags in UTCB
    - but remember it's asynchronous and can change at any time
  - 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)

Be open.
Be safe.

- 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 MemoryCopy() syscall
- 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
- 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

Open Kernel Labs™

Be open.
Be safe.

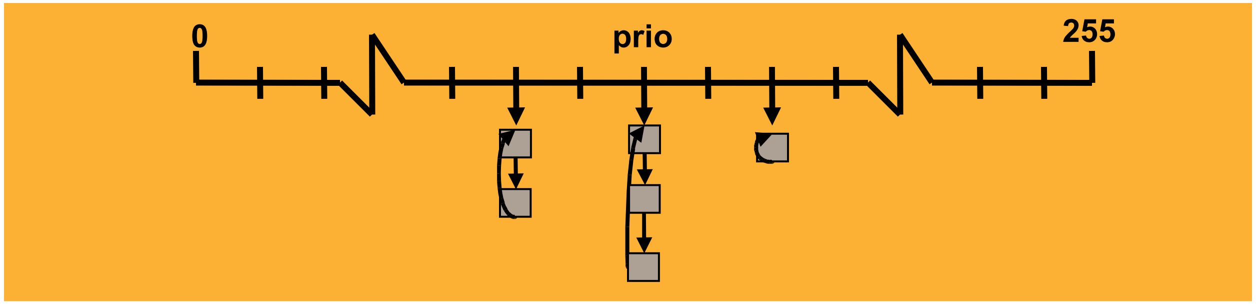
→ Resource-control system calls
- ThreadControl
- SpaceControl
- MapControl
- CapControl
- MutexControl
- InterruptControl
- CacheControl
- PlatformControl

→ Other system calls
- ExchangeRegisters
- Ipc
- *Schedule*
- ThreadSwitch
- Mutex
- MemoryCopy
- SpaceSwitch

→ 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 prio
- Aim is *real-time scheduling*, **not** fairness
  - Kernel on its own will *never* change the priority of a thread
  - Achieving fairness (if desired) is the job of user-level servers
  - Can use Schedule() syscall to adjust priorities



OKL4 Scheduling

Be open.
Be safe.

- 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 the caller or unblocks another thread
    - but see below for exceptions...
- Scheduler is **not** invoked when:
  - Interrupt occurs
    - Makes interrupt handler thread runnable
    - Thread to run is determined by priorities of current and interrupt thread
  - The highest-prio thread can be determined without the scheduler
    - eg. send unblocks other thread ⇒ run sender or receiver based on prio
    - switch without scheduler invocation is called *direct process switch*
- Kernel optionally implements *schedule inheritance*
  - if high-prio thread is blocked on other thread (through IPC, mutex)

Schedule

- → The Schedule() syscall does **not** invoke a scheduler!
  - … nor does it actually schedule any threads
- → Schedule sets/reads a thread's scheduling parameters
  - 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:**

```
int  status;
status = L4_Set_Priority (thread, prio);
printf("%Id\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)
  - 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

OKL4 API Overview

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - *ThreadSwitch*
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

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

Open Kernel Labs™

Be open.
Be safe.

- Directed switch (time-slice donation):

```
L4_ThreadSwitch (thread);
```

- Yield (undirected switch):

```
L4_Yield ();
```

OKL4 API Overview

Be open.
Be safe.

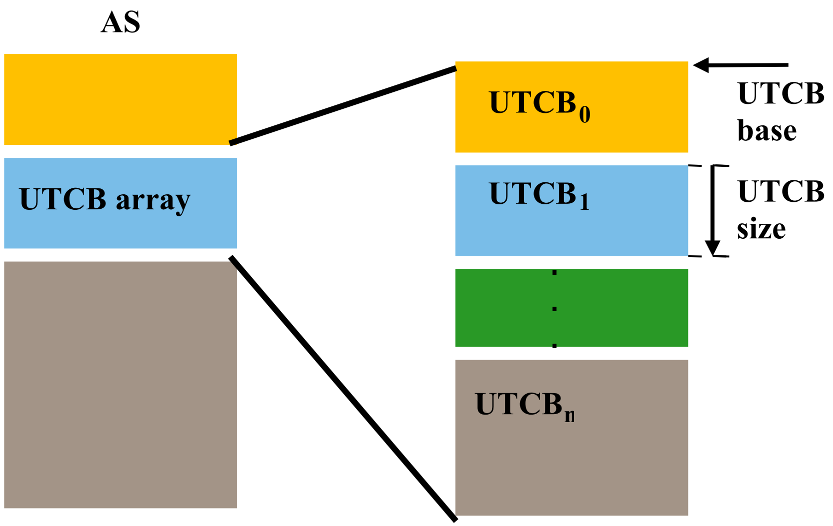
→ Resource-control system calls
- ThreadControl
- *SpaceControl*
- MapControl
- CapControl
- MutexControl
- InterruptControl
- CacheControl
- PlatformControl

→ Other 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*
    - 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

```
L4_Word_t L4_SpaceControl( L4_SpaceId_t   target,
                           L4_Word_t      control,
                           L4_ClistId_t   clist,
                           L4_Fpage_t     utcb_region,
                           L4_Word_t      resources,
                           L4_Word_t      *old_resources);
```

- Can modify address-space resources (hardware-dependent)
  - MMU ASID
  - ARM9 PID
- Note: Should use libl4 APIs rather than raw system call
  - See discussion of tasks for usage examples
- *OKL4 2.1 space pager has been removed*

Deleting Address Spaces

Be open.
Be safe.

- 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

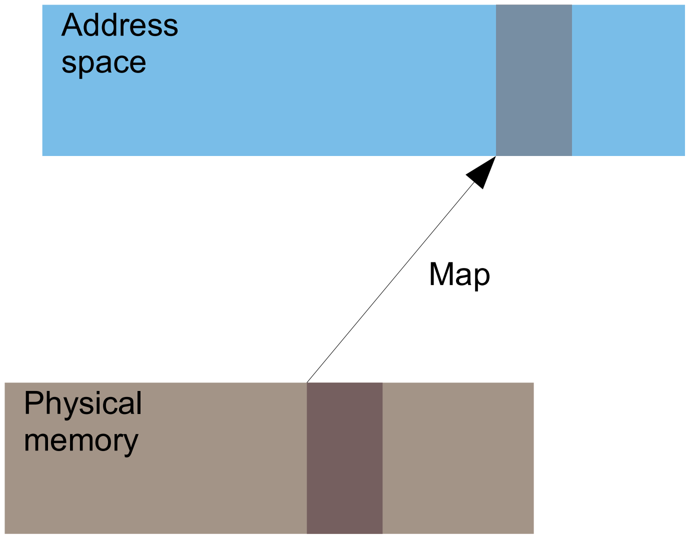
- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - *MapControl*
  - CapControl
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

Address Spaces

Open Kernel Labs™

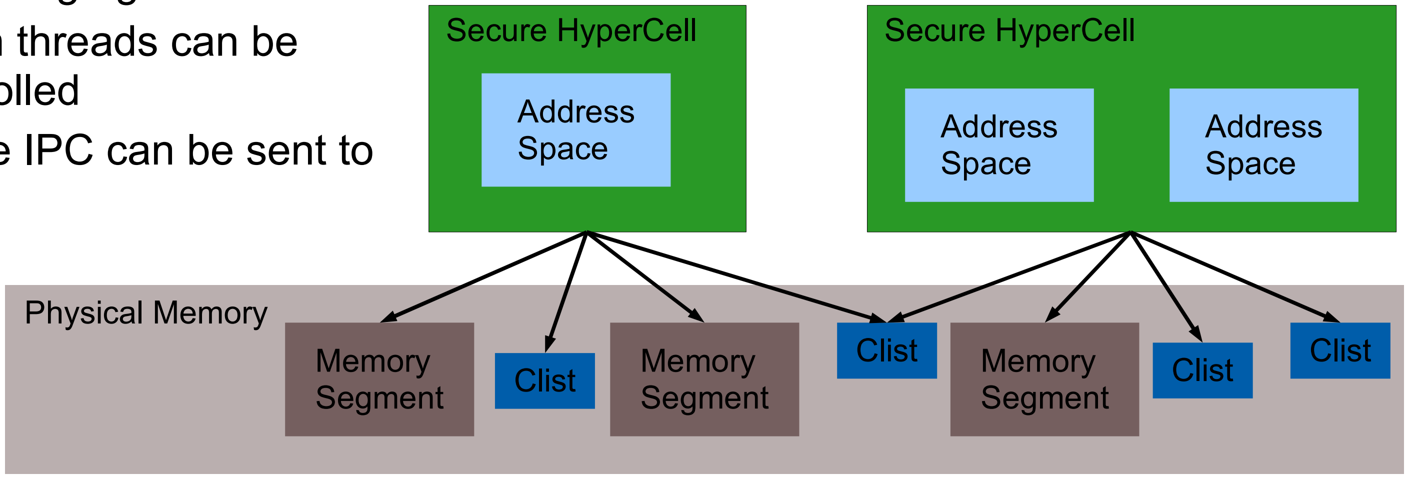
Be open.
Be safe.

- Address spaces are created empty
  - Except for UTCB
- Need to be explicitly populated with page mappings
  - Kernel does not map pages automatically (except for UTCB)
- Normally address space populated by pager on demand
  - Thread runs, faults on unmapped pages, pager creates mapping
- OS server(s) can populate pro-actively
  - Startup programs from the boot image get their executable and initialized data pre-mapped



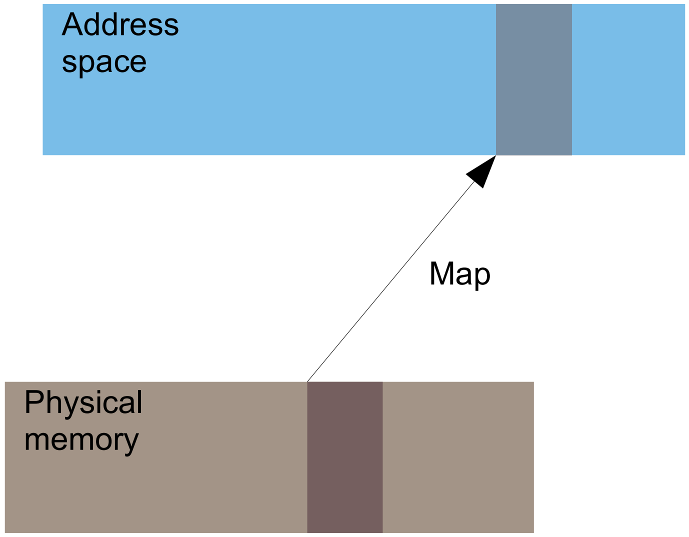
Secure HyperCells

- A *Secure HyperCell*™ (SHC) is an isolation domain
  - consists of one or more address spaces
  - all sharing the same resource rights
- SHCs are defined (statically) at system-configuration time (Elfweaver)
- *Future releases of OKL4 will provide more dynamic rights management*
- A SHC has
  - *memory segments*, defining rights to physical memory
    - which memory can be mapped into address spaces
  - *clists*, defining rights on threads
    - which threads can be controlled
    - where IPC can be sent to



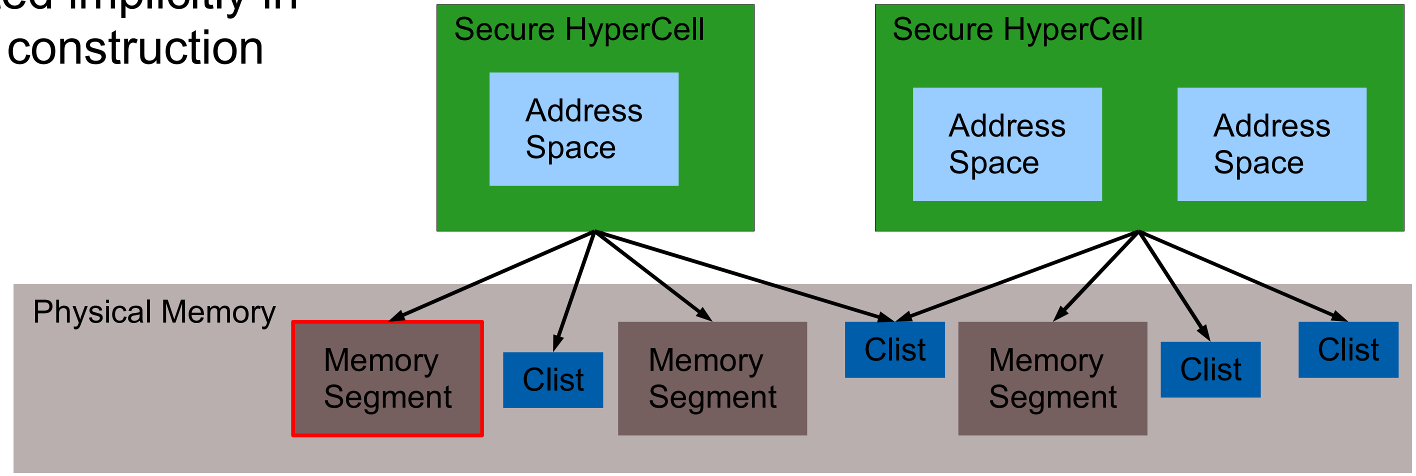
MapControl

- Creates (maps), destroys (un-maps) or changes page mappings
- Requires access to a physical memory segment
- Single MapControl call can create/destroy/modify multiple mappings
  - individual mappings described by *map items*
- Each map item describes a mapping
  - can be multiple pages
  - consists of:
    - segment descriptor
      - indicates mapping source
      - physical memory address
    - virtual descriptor
      - indicates mapping destination
      - virtual memory address
    - size descriptor



Physical-Memory Segments

- Memory segment refers to contiguous range of physical memory
- Identified by a unique *segment ID*
  - defined in Elfweaver input
- Allocation of a segment to a SHC implies the right to map the segment
- Segment descriptor in map item contains
  - segment ID
  - offset from segment start
- Constructed implicitly in map item construction



Specifying Mappings: Map Items

Be open.
Be safe.

```
void L4_MapItem_Map(L4_MapItem_t *m,          /* new map item */
                    L4_Word_t     segment_id, /* phys memory */
                    L4_Word_t     offset,     /* source address */
                    L4_Word_t     vaddr,      /* destination addr */
                    L4_Word_t     page_size,
                    L4_Word_t     attr,
                    L4_Word_t     rwx);
```

- Map one page of specified size
  - from physical address = segment base-address + offset
  - to virtual address = vaddr
  - caching attributes specified by attr
  - page permissions specified by rwx
- Similar APIs for modifying or removing mappings
- Future versions of OKL4 will support multi-page map items

Establish a Mapping

```
L4_Word_t ProcessMapItems( L4_SpaceId_t s,    /* destin. space */
                           L4_Word_t    n,    /* nbr map items */
                           L4_MapItem_t *m);  /* map item array */
```

→ Calls MapControl() to establish mappings in destination address space
  - processes each map item in one system call
  - can be a combination of map/unmap/modify

Superpages

Be open.
Be safe.

- Many processors support several page sizes
  - OKL4 usually supports all sizes supported by hardware
  - actually supported sizes can be obtained from libl4
- Page size to be used in mapping is specified in map item
  - must be supported by MMU
  - virtual and physical addresses in mapping must be aligned to this page size
- *Prior to OKL4 3.0, kernel chose largest possible page size for mappings*
  - this is policy that has been removed from the 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 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 tit runnable
  - first thread may start any further threads itself

Creating a Task…

Be open.
Be safe.

→ Define UTCB area location in new address space

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

Adding Memory to a Task

Be open.
Be safe.

- Set up map items:

```
m = &map_items;
L4_MapItem_Map( map_item[n++], img_seg,
                text_off, text_virt,
                page_sz, L4_CachedMemory,
                L4_eXecutable);
...
L4_MapItem_Map( map_item[n++], img_seg,
                data_off, data_virt,
                page_sz, L4_CachedMemory,
                L4_ReadWriteOnly);
...
```

- Process mappings:

```
L4_ProcessMapItems (task, n, map_items);
```

Adding Threads to Task

Open Kernel Labs™

Be open.
Be safe.

- Use ThreadControl() to add new threads to AS

```
threadno = …;                    /* according to policy */
thread   = L4_CapId (TYPE_CAP, threadno);
utcb     = …;                    /* according to policy, ignore on ARM9/11 */
L4_ThreadControl (tid, task, me, pager, me, res, utcb);
```

- Note: Maximum number of threads defined at address-space creation time
  - via the size of the UTCB area

Starting first Thread in Task

Be open.
Be safe.

```
L4_Start_SpIp (thread, stack_pointer, function);
```

- convenience function, uses the ExchangeRegisters() system call
- thread can be started by its pager
- alternatively, can be started by another thread in the same address space

Practical Considerations

Be open.
Be safe.

- → 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 IDL compiler (Magpie)
- → Note: *Some older L4 APIs do not support space IDs*

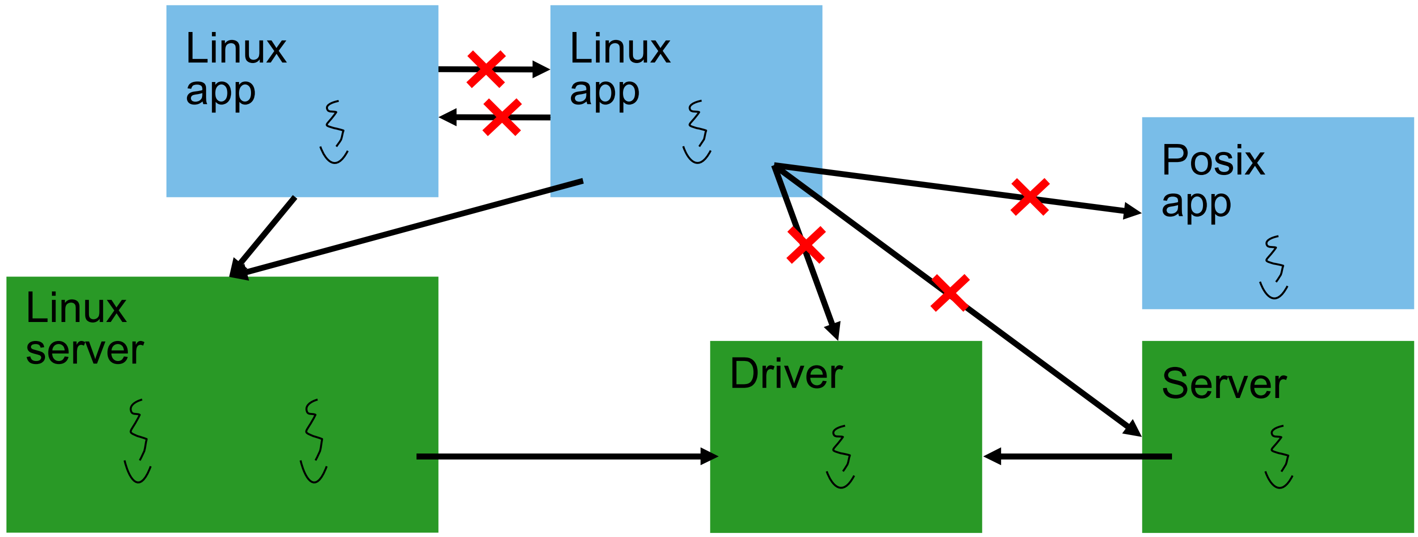
OKL4 API Overview

Be open.
Be safe.

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - *CapControl*
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

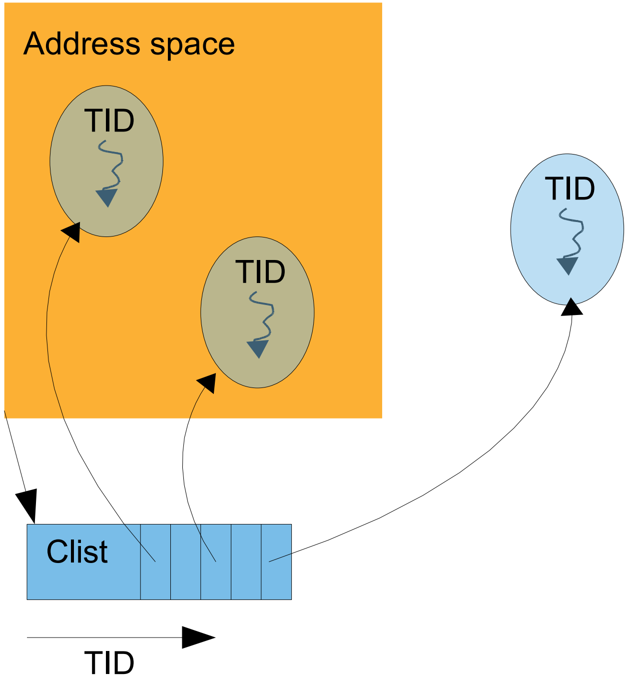
Capabilities

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



Capabilities and Clists

- Each address space has one *clist*
  - clist belongs to the Secure HyperCell of the address space
  - holds all caps of that address space (except reply caps)
  - caps provide local names for kernel objects
- Clist is a kernel object
  - not directly accessible to user code
- *Clists* created/destroyed 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

```
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 libl4 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

Be open.
Be safe.

→ Resource-control system calls

- ThreadControl
- SpaceControl
- MapControl
- CapControl
- *MutexControl*
- InterruptControl
- CacheControl
- PlatformControl

→ Other system calls

- ExchangeRegisters
- Ipc
- Schedule
- ThreadSwitch
- *Mutex*
- MemoryCopy
- SpaceSwitch

→ Protocols

- Page fault
- Exception
- Interrupt

Mutex

Be open.
Be safe.

- Kernel-supported mutual-exclusion mechanism
- Two kind 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 uncontented
    - syscall if contented
    - 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

Open Kernel Labs™

Be open.
Be safe.

MutexControl() convenience functions:

```
okl4_mutex_t mutex;
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);
ok = okl4_mutex_trylock(  mutex);
ok = 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

OKL4 API Overview

Open Kernel Labs™

Be open. Be safe.

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - *MemoryCopy*
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

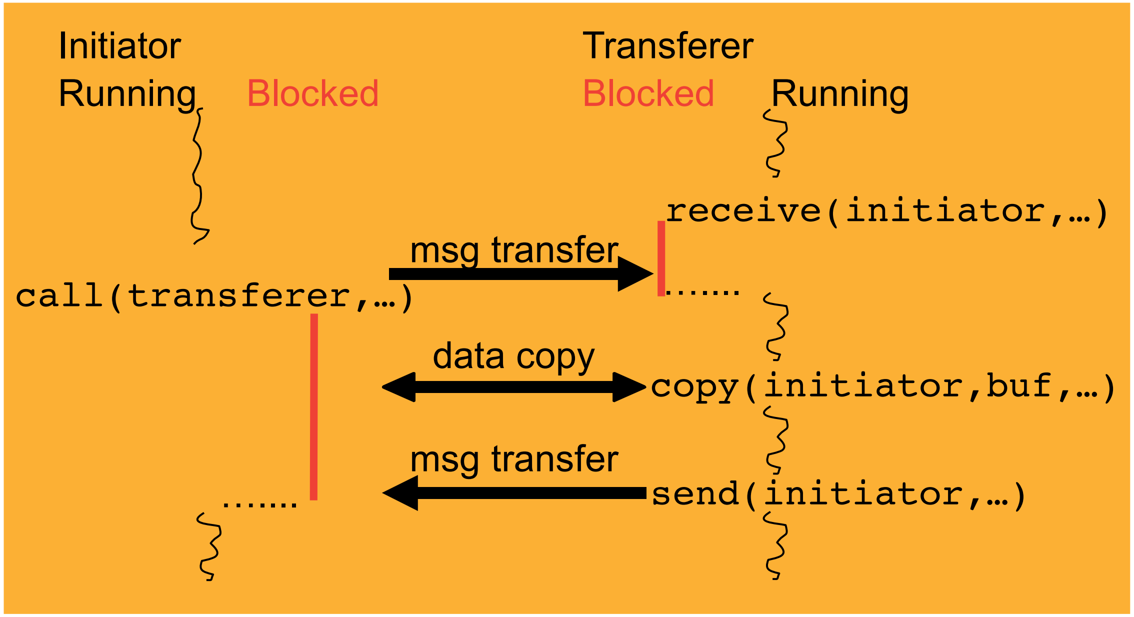
User-to-User Memory Copy

Be open.
Be safe.

- 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
  - new in OKL4 2.1
  - 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
    - setup
    - synchronization
  - don't touch buffer
    - sender between IPCs
    - receiver after final IPC
- Copy direction
  - independent of IPC direction



MemoryCopy Use

Be open.
Be safe.

- Initiator:

```
L4_MsgClear(        &msg);
L4_Set_MemoryCopy( &msg);                /* Set m bit in tag word */
L4_MsgAppendWord(  &msg, &buf);          /* buffer address */
L4_MsgAppendWord(  &msg, n);             /* buffer size (byte) */
L4_MsgAppendWord(  &msg, L4_MemoryCopyBoth<<30); /* 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_MemoryCopy (initiator, &rec_buf, n_rec, L4_MemoryCopyFrom);
L4_MemoryCopy (initiator, &snd_buf, n_snd, L4_MemoryCopyTo);
```

OKL4 API Overview

Open Kernel Labs™
Be open.
Be safe.

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - *SpaceSwitch*
- Protocols
  - Page fault
  - Exception
  - Interrupt

SpaceSwitch

Be open.
Be safe.

- Migrates a thread between address spaces
  - previously part of ThreadControl() functionality
- Requires special privilege associated with address spaces involved
  - “kernel resource object”
  - needed for caller, source and destination
  - allocated at system-configuration time
- Useful in some special cases, *generally best to avoid*
- *Introduced* in OKL4 2.1

OKL4 API Overview

Open Kernel Labs™

Be open.
Be safe.

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - *InterruptControl*
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

InterruptControl

- Manages interrupts
  - association/dissociation of handler threads with IRQs
  - interrupt acknowledgement
  - *right to use is granted at system-configuration time*
  - 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**: ack by a different thread in same address space
  - **acknowledgeWait**: ack and wait for next interrupt notification
- *Introduced in OKL4 2.1*

InterruptControl Use

```
L4_Register_Interrupt(
        L4_CapId_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 msg 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 MR_0 ... MR_{mrs}
  - typically mrs=0, parameter word contains IRQ number
  - additional request parameter not needed by most platforms

OKL4 API Overview

Open Kernel Labs™

Be open.
Be safe.

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - *CacheControl*
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

CacheControl

Be open.
Be safe.

- 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)
```

OKL4 API Overview

Be open.
Be safe.

→ Resource-control system calls
- ThreadControl
- SpaceControl
- MapControl
- CapControl
- MutexControl
- InterruptControl
- CacheControl
- *PlatformControl*

→ Other system calls
- ExchangeRegisters
- Ipc
- Schedule
- ThreadSwitch
- Mutex
- MemoryCopy
- SpaceSwitch

→ Protocols
- Page fault
- Exception
- Interrupt

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 is granted at system-configuration time*
- Possible use:

```
L4_PlatformControl(0, bus_freq, cpu_freq, voltage);
```

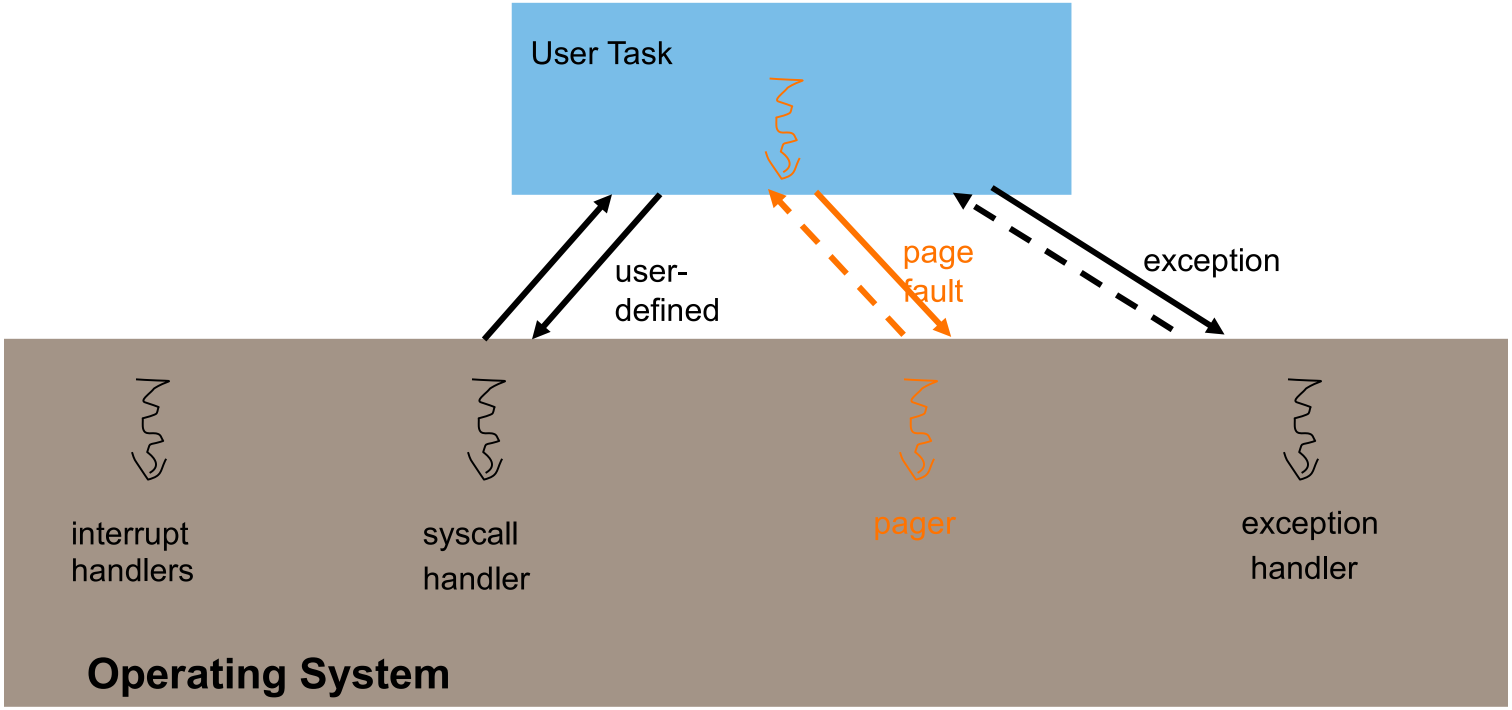
OKL4 API Overview

Be open.
Be safe.

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt

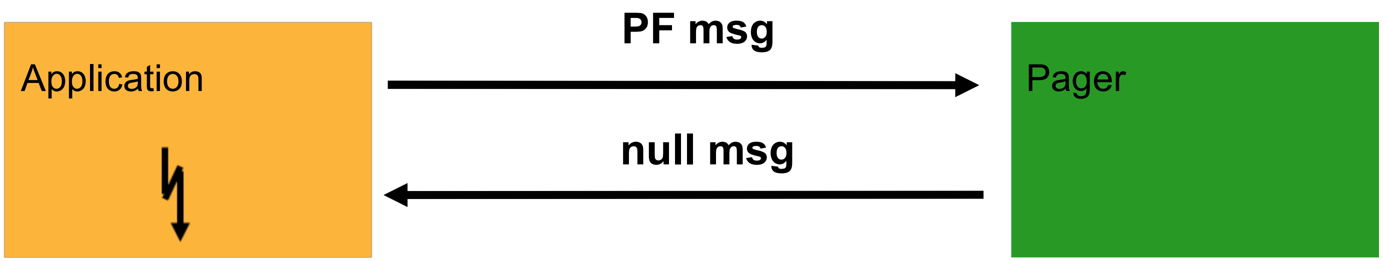
OKL4 Protocols

- *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

Be open.
Be safe.

- Format of kernel-generated page fault message

Fault IP					MR 2
Fault address					MR 1
-2	0rwx	$0_{(4)}$	$0_{(6)}$	2	MR 0

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

Fault IP					MR 2
0x2002					MR 1
-2	0rwx	$0_{(4)}$	0	2	MR 0

- 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

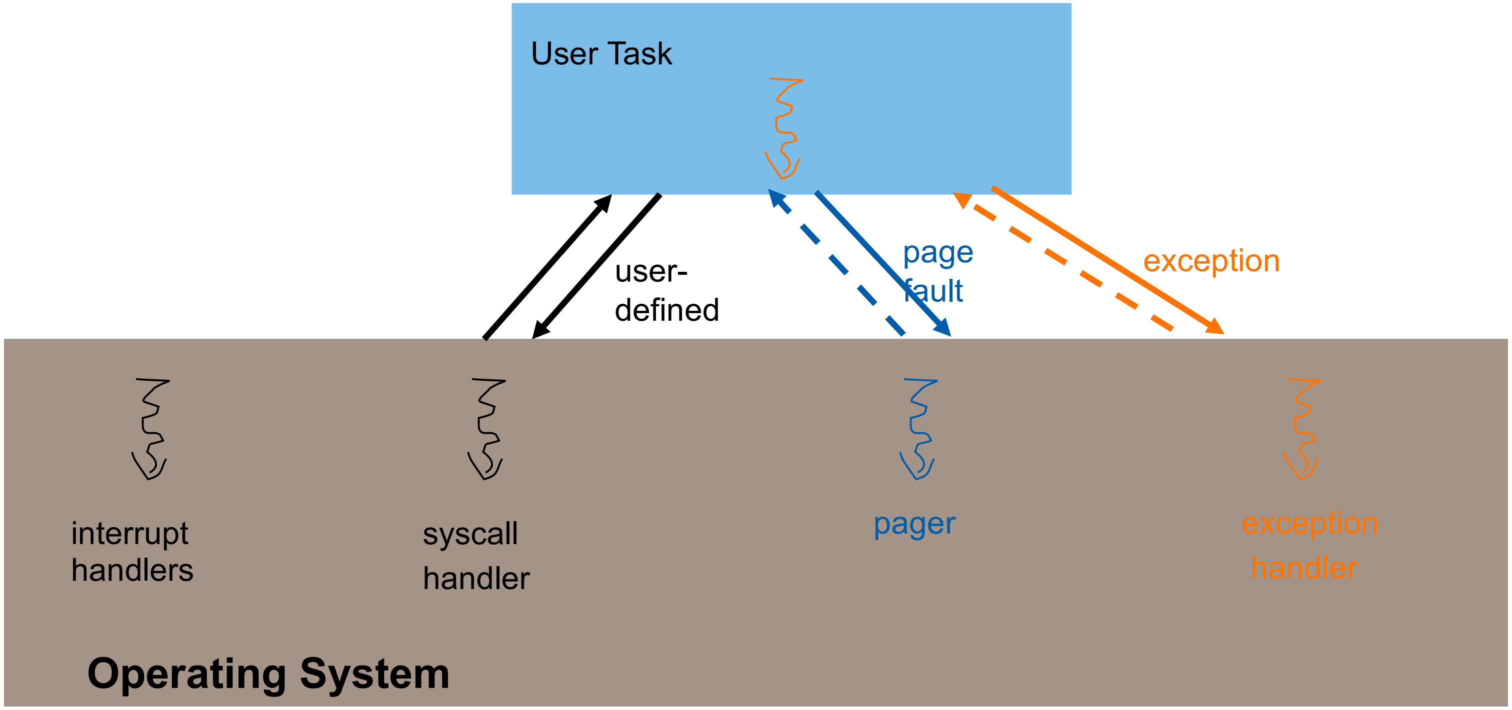
0x8	12	0

0x300	0

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

OKL4 Protocols

- Page fault
- *Exception*
- Interrupt



Exception Protocols

Be open.
Be safe.

- 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$_{k-1}$					MR $_{k+1}$
Exception word$_0$					MR $_2$
Exception IP					MR $_1$
label	0	0	0	k	MR $_0$

- 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 (eg in status register)

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

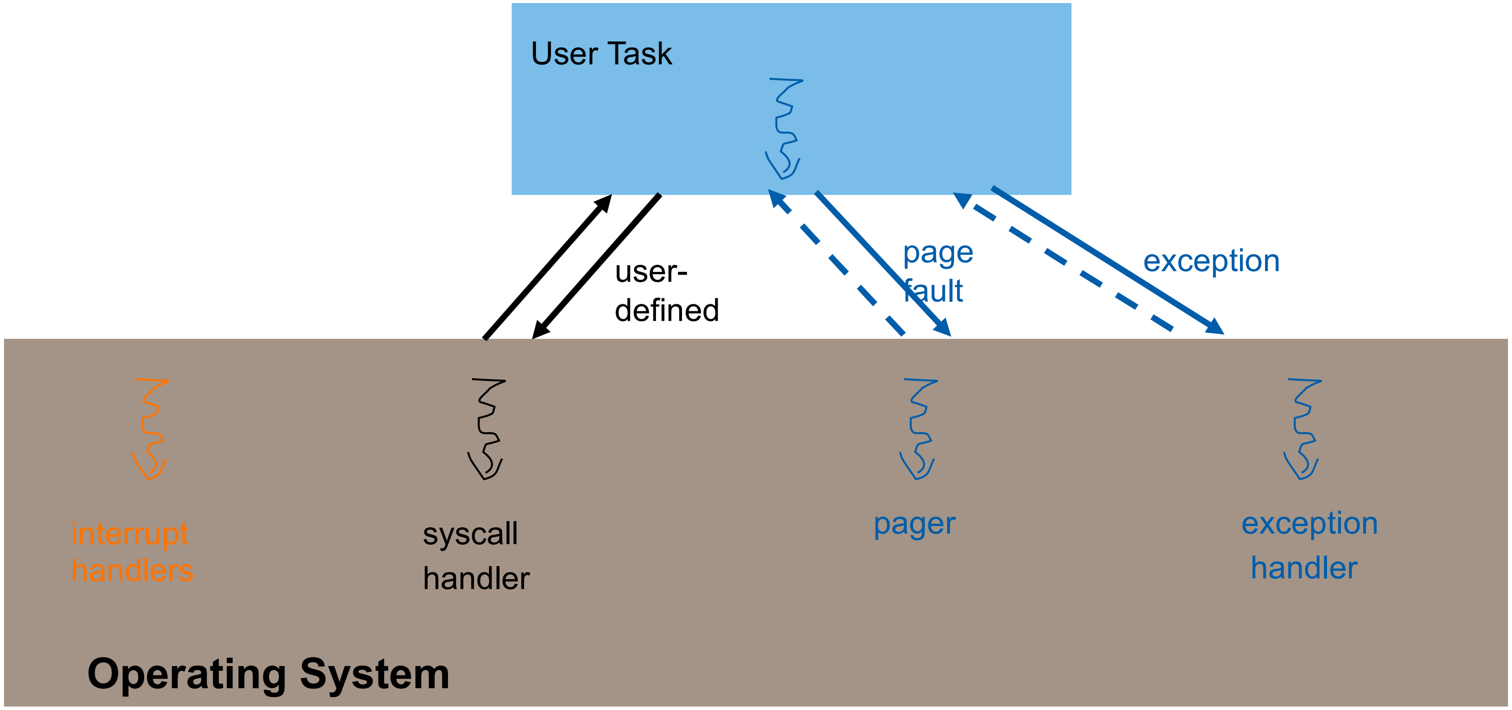
handler: reply with IP of local exception handler code to be executed by the thread itself

ignore: will block the thread indefinitely

kill: use ExchangeRegisters() (if local) or ThreadControl() to restart or kill thread

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

Be open.
Be safe.

- Memory mapped
  - Mapped into the hardware address space
  - Can be access using normal memory operations
    - map into driver address space using MapControl()
    - 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

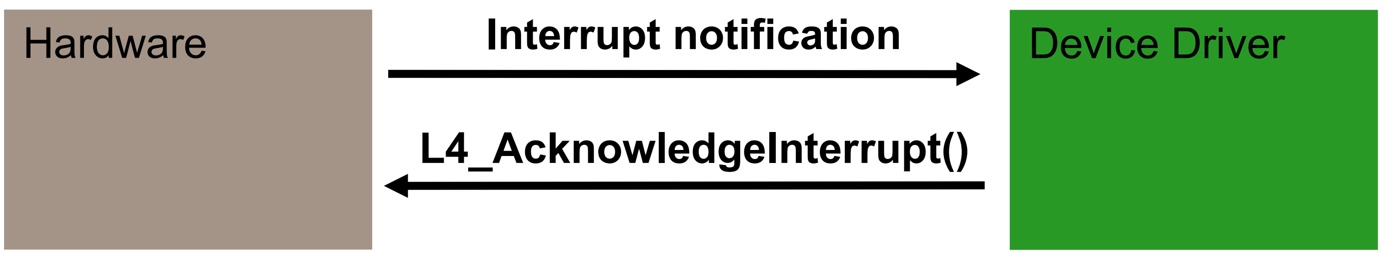
Interrupts

Be open.
Be safe.

- 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

Be open.
Be safe.

- 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

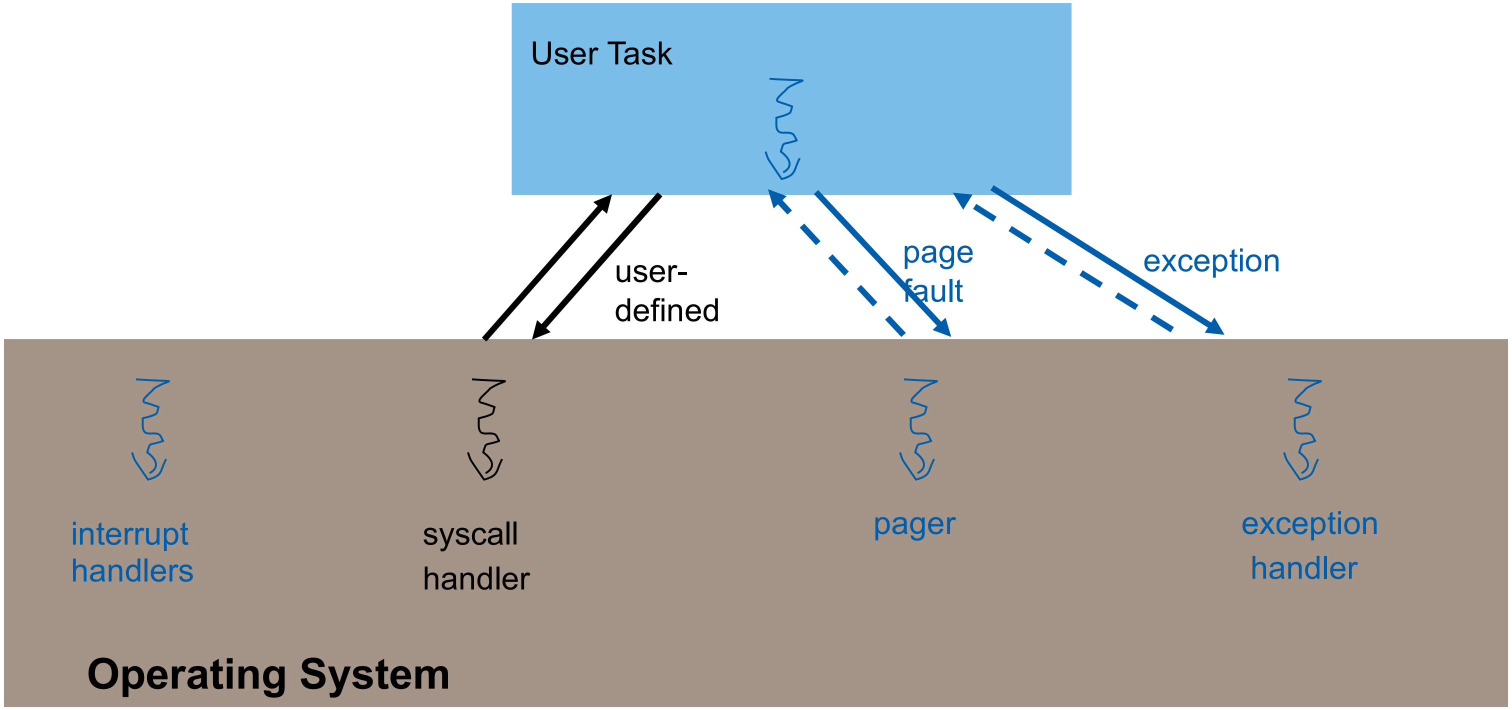
Open Kernel Labs™

Be open.
Be safe.

- DMA is important for performance of bulk-IO 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
  - requires OS server to keep track of these mappings

OKL4 Protocols

- Page fault
- Exception
- Interrupt



OKL4 API Overview

Be open.
Be safe.

- Resource-control system calls
  - ThreadControl
  - SpaceControl
  - MapControl
  - CapControl
  - MutexControl
  - InterruptControl
  - CacheControl
  - PlatformControl
- Other system calls
  - ExchangeRegisters
  - Ipc
  - Schedule
  - ThreadSwitch
  - Mutex
  - MemoryCopy
  - SpaceSwitch
- Protocols
  - Page fault
  - Exception
  - Interrupt