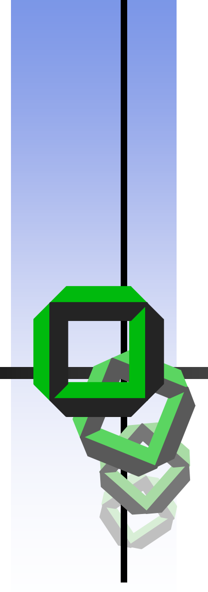


Systems Design and Implementation

II.1 – L4 API Crash Course Part I

System Architecture Group, SS 2007

University of Karlsruhe

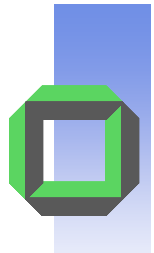
22 April 2009

Jan Stoess

University of Karlsruhe

Tuesdays 17:30-19:00 SR-134, 50.41 (AVG)
Thursdays 15:45-17:15 SR-134, 50.41 (AVG)

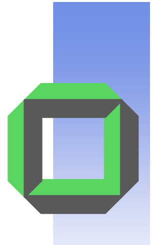
Based on slides by Jochen Liedtke and Kevin Elphinstone



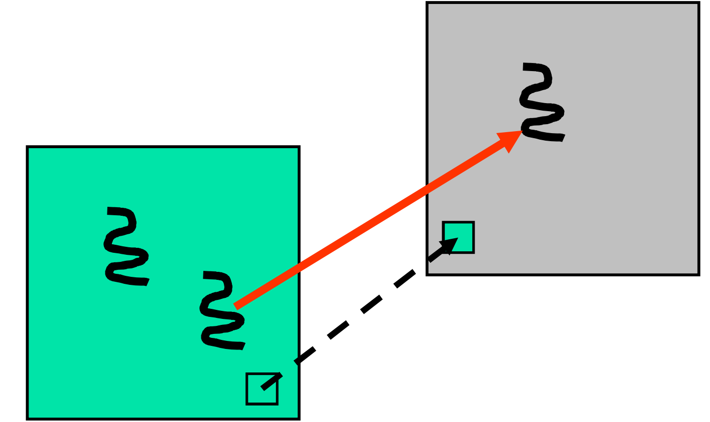
L4 X.2 API Reference Manual

- Available from http://l4ka.org/
  - Latest version always in the news box on the right
- Defines the kernel API + ABI
  - System call semantics and parameters
  - C++ style API definition
    - Data types
    - Header file to include
    - Generic programming interface
    - Convenience programming interface
    - Support functions
  - Binary interface for supported architectures
- Does not describe how to *use* the kernel

© 2009 University of Karlsruhe, System Architecture Group

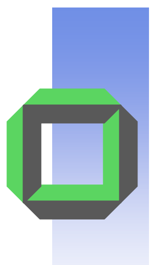


Fundamental L4 Concepts



- Two abstractions
  - Address Spaces
    - Units of protection
    - Resource management
  - Threads
    - Execution entities
    - Carry unique identifiers
- Two mechanisms
  - Communication – IPC
    - Synchronous, between threads
    - Identification: thread ids
  - Rights delegation – Mapping
    - Address space construction via IPC
    - FlexPages
      - Architecture independent page abstraction
      - Describe range of virtual address space

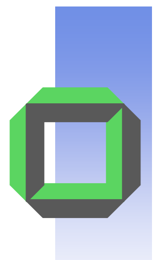
© 2009 University of Karlsruhe, System Architecture Group



Fundamental L4 Concepts

- User-level pagers
  - Kernel turns page faults into IPC message
  - Establish mapping in reply
- User-level device drivers
  - Device drivers run as unprivileged user threads
  - Hardware interrupts are delivered via IPC
    - Unless used by the kernel internally, e.g. timer interrupt
  - Acknowledge interrupt in reply
- User-level exception handlers
  - Exceptions are delivered via IPC
    - Unless used by the kernel internally, e.g. FPU virtualization
  - Fix exception cause or modify faulting thread in reply
- Goal: No policy in kernel
  - Makes kernel universal

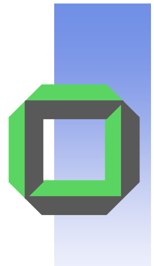
© 2009 University of Karlsruhe, System Architecture Group



Microkernel System Calls

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

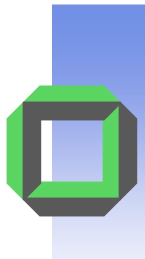
© 2009 University of Karlsruhe, System Architecture Group



Initial Servers

- Created by kernel at boot time
  - Sigma0
    - Initial address space
    - Root of all mappings
    - “Owns” all physical memory
  - Root task
    - First freely usable user thread
    - Address space backed by sigma0
- Can perform privileged system calls
  - ThreadControl
  - SpaceControl
  - ProcessorControl
  - MemoryControl

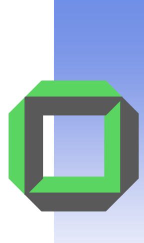
© 2009 University of Karlsruhe, System Architecture Group



Kernel Interface Page

- Kernel memory object in the address space of a task
  - Placed on address space creation
    - Location dictated by SpaceControl system call
  - No page faults on access
- Contains information about the kernel and the machine
  - API version, kernel features
  - Kernel system call entry points
  - Supported page sizes
  - Format and number of thread IDs
  - Memory layout – Physical memory, virtual address space
  - Processors – core speed, bus speed
  - …

© 2009 University of Karlsruhe, System Architecture Group



KernelInterface

- Locates the kernel interface page
- Special system call
  - Illegal instruction on x86 – why?
  - Doesn't use KIP for calling
  - Slow system call (expensive)

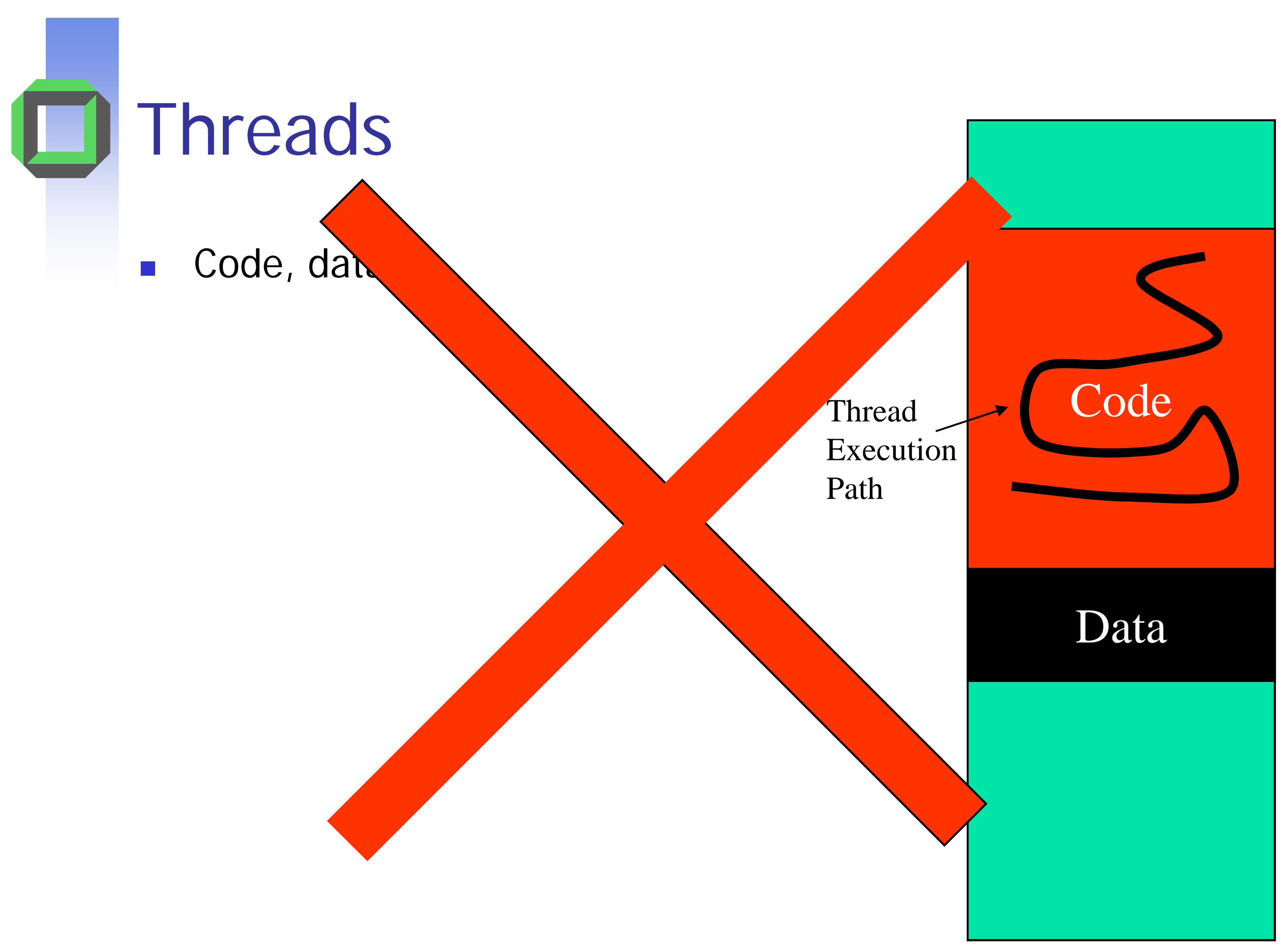
```
void * L4_KernelInterface (L4_Word_t *ApiVersion,
                           L4_Word_t *ApiFlags,
                           L4_Word_t *KernelId)
```

- Returns
  - Pointer to KIP
  - API version and flags – revision, word width, endianess
  - Kernel Id – identifies implementation

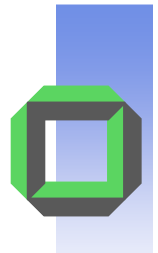
© 2009 University of Karlsruhe, System Architecture Group

Threads

- Code, dat



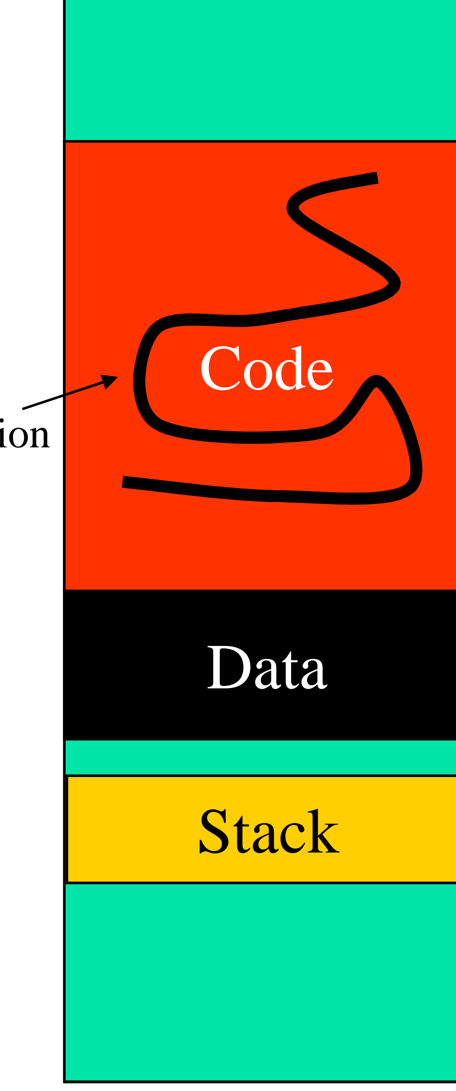
© 2009 University of Karlsruhe, System Architecture Group



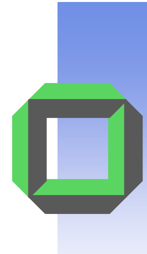
Traditional Thread

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

Thread Execution Path



© 2009 University of Karlsruhe, System Architecture Group

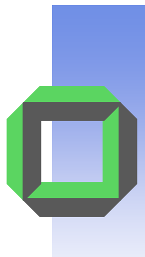


L4 Thread = Thread + …

- A set of (virtual) registers and – see next slide
- A priority and a timeslice
- A unique thread identifier
- An associated address space

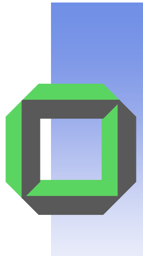
- L4 provides a fixed number of threads in the entire system
  - Root task responsible for creating/deleting threads and assigning them to address spaces
  - System, User and “Hardware” threads

© 2009 University of Karlsruhe, System Architecture Group



Virtual Registers

- Per-thread “register set” defined by the microkernel
- Map to real machine registers or memory locations
  - Mapping depends on architecture and ABI
    - IA-32: 1-3 virtual registers in GPRs, others in memory
    - IA-64: 8 in GPRs
- Three basic types
  - Thread Control Registers (TCRs)
    - Share information about threads between kernel and user level
  - Message Registers (MRs)
    - Contain the message (or description of it, e.g. region of memory)
  - Buffer Registers (BRs)
    - Specify where complex message parts are received

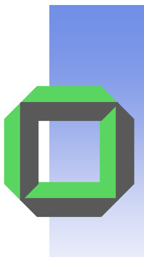


Thread Control Blocks (TCBs)

- State of a thread is stored in its thread control block
- Security considerations
  - Some state can only be modified via a controlled interface (system calls)
    e.g., address space associated with the thread
  - Other state can be freely accessible by user-level applications without compromising the system
    e.g., pager thread associated with the thread

- Put uncritical state in a user-level TCB (UTCB)
  - more efficient access

© 2009 University of Karlsruhe, System Architecture Group

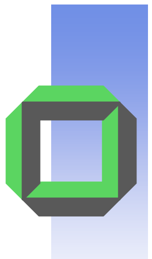


Thread Control Registers

- Stored in UTCB
  - Pinned memory, no page faults on access
  - UTCB area dictated at address space creation using SpaceControl
  - UTCBs assigned via ThreadControl
- Never access them directly
  - Only modified via provided programming interface
- Most TCRs are set/read in the context of other actions (e.g. IPC)

<table>
<tr><td colspan="3">ThreadWord1</td></tr>
<tr><td colspan="3">ThreadWord0</td></tr>
<tr><td colspan="3">Virtual/ActualSender (rw, IPC)</td></tr>
<tr><td colspan="3">IntendedReceiver (ro, IPC)</td></tr>
<tr><td colspan="3">ErrorCode (ro, IPC)</td></tr>
<tr><td colspan="3">XferTimeouts (rw, IPC)</td></tr>
<tr><td>~</td><td>Cop (wo)</td><td>Preempt (rw)</td></tr>
<tr><td colspan="3">ExceptionHandler (rw)</td></tr>
<tr><td colspan="3">Pager (rw, VM)</td></tr>
<tr><td colspan="3">UserDefinedHandle (rw, Threads)</td></tr>
<tr><td colspan="3">ProcessorNo (ro)</td></tr>
<tr><td colspan="3">MyGlobalId (ro, Threads & IPC)</td></tr>
</table>

© 2009 University of Karlsruhe, System Architecture Group



Thread Identifiers

- Global Identifiers
  - Identify a thread uniquely within the system
  - No policy – freely assignable
- Local Identifiers
  - Identify a thread within an address space
    - Unique and useable only within an address space
    - Typically the address of the thread's UTCB
- Can translate one to another
- Special Identifiers
  - nilthread – no thread
  - anythread – wildcard

Global Thread Id

Thread No (18)	Version (14)

Global Interrupt Id

Interrupt No (18)	1 (14)

Local Thread Id

Local Id/64 (26)	000000

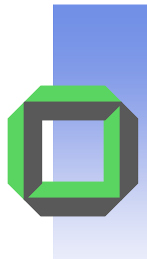
nilthread

0 (32)

anythread

-1 (32)

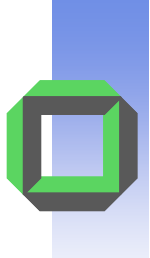
© 2009 University of Karlsruhe, System Architecture Group



ThreadControl

- Create, destroy, or modify threads
- Determines a thread's
  - Global thread identifier
  - Address space it executes in
  - Scheduler (thread permitted to control scheduling parameters)
  - Pager (thread that receives page fault messages)
  - Location of the UTCB within the address space's allotted UTCB area (See SpaceControl later)
- Threads can be created *active* or *inactive*
  - *Inactive*
    - Create and manipulate a new address space
    - Allocate a new thread to an existing address space

© 2009 University of Karlsruhe, System Architecture Group



ThreadControl

```
L4_Word_t L4_ThreadControl (L4_ThreadId_t dest,
                            L4_ThreadId_t SpaceSpecifier,
                            L4_ThreadId_t Scheduler,
                            L4_ThreadId_t Pager,
                            void * UtcbLocation)
```

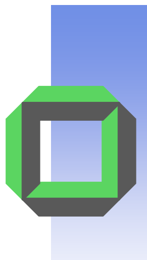
- **`SpaceSpecifier != dest`**
  Creates thread **`dest`** in the address space of thread **`SpaceSpecifier`**
  Note: implicit naming of address spaces

- **`SpaceSpecifier == dest`**
  Creates thread **`dest`** in its own (new) address space

- **`SpaceSpecifier == nilthread`**
  Deletes existing thread **`dest`**

- **`pager == nilthread`**
  Inactive, otherwise active

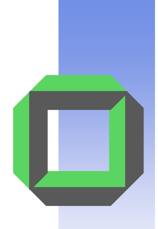
© 2009 University of Karlsruhe, System Architecture Group



Steps in Creating a New “Task”

- Task = Address Space + Thread
- A task has
  - Thread state
    - Identifier, IP, SP, pager, scheduler, UTCB location
  - Address space state
    - UTCB area, kernel interface page area, redirector
  - Code, data, and stack mapped to address space

© 2009 University of Karlsruhe, System Architecture Group

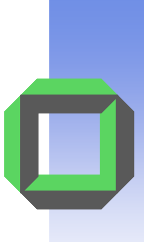


Steps in Creating a New “Task”

1. Create an inactive thread in a new address space.

```
L4_ThreadControl (
      task,              /* new tid */
      task,              /* new space identifier */
      me,                /* scheduler of new thread */
      L4_nilthread,      /* pager = nil, inactive */
      (void *) -1        /* NOP Utcb location */
);
```

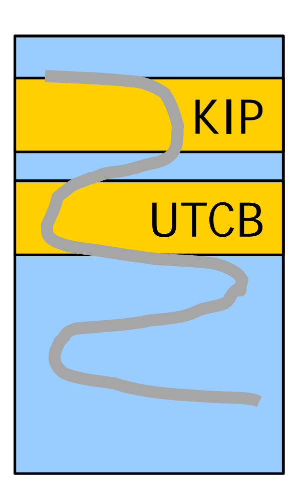
© 2009 University of Karlsruhe, System Architecture Group



Steps in Creating a New "Task"

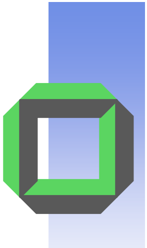
2. Set location of KIP and UTCB area in the new address space.

```
L4_SpaceControl (
        task,
        0,                /* control (ignore) */
        kip_area,
        utcb_area,
        L4_anythread,     /* redirector */
        &control          /* output (ignore)  */
);
```



- `kip_area` and `utcb_area` are flexpage descriptors
- `redirector = anythread`
  - Threads in the space can talk to all other threads
- `control` is an architecture-specific parameter, ignore for now

© 2009 University of Karlsruhe, System Architecture Group



Steps in Creating a New "Task"

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

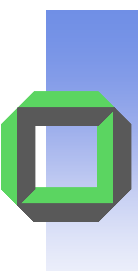
```
L4_ThreadControl (
        task, task, me,
        pager,                  /* new pager */
        (void *) utcb_base      /* utcb location */
);
```

KIP

UTCB

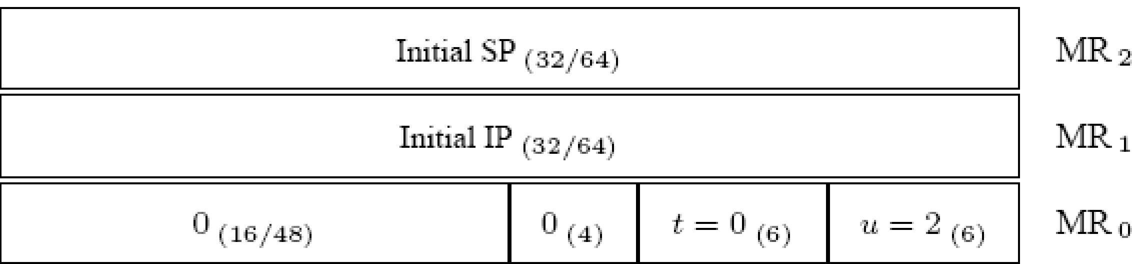
- The thread will wait for an IPC from the pager.
- The message must contain the IP and SP of the new thread.

© 2009 University of Karlsruhe, System Architecture Group



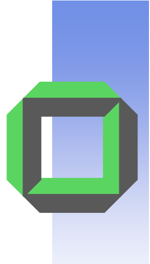
Steps in Creating a New "Task"

4. Send an IPC to the new thread with the IP and SP in the first two words of the message.



- The thread will start executing at the received IP with the SP set as received.

© 2009 University of Karlsruhe, System Architecture Group



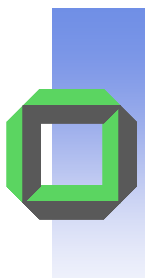
Adding extra inactive threads to a task

- Use ThreadControl to assign new inactive threads to an existing address space

```
L4_ThreadControl (
        newtid,          /* new thread id */
         ExistingId,     /* address space identifier */
         me,             /* scheduler of new thread */
         L4_nilthread, /* pager = nil, inactive */
         (void *) -1     /* NOP Utcb location */
);
```

- Note: Can also add active threads

© 2009 University of Karlsruhe, System Architecture Group

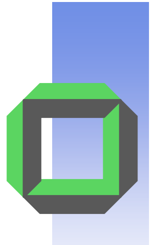


ExchangeRegisters

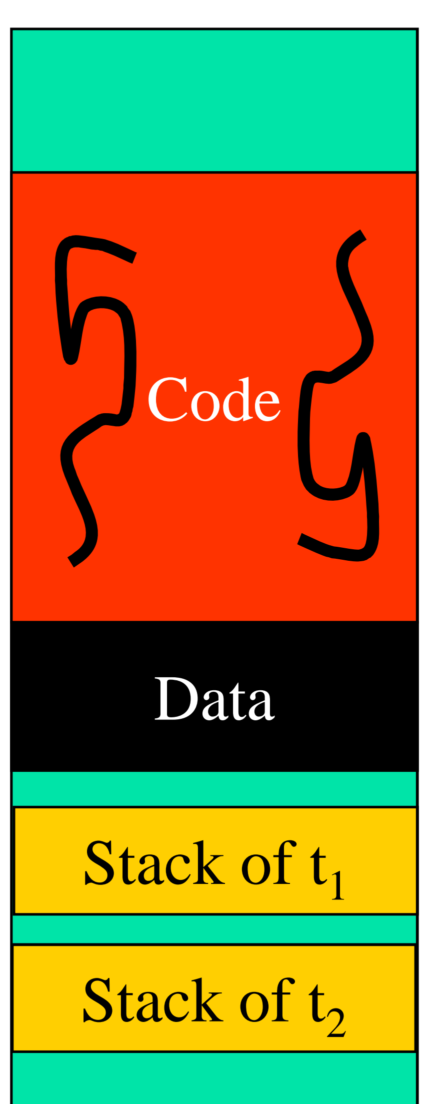
- Manipulating threads within an AS
  - IP, SP
  - User-defined handle
  - Pager
- Suspend/resume (i.e. activate/deactivate)
- Convert thread IDs – local ↔ global

```
L4_ThreadId_t L4_ExchangeRegisters (
        L4_ThreadId_t dest,
        L4_Word_t control,
        L4_Word_t sp, L4_Word_t ip,
        L4_Word_t flags, L4_Word_t UserDefHandle,
        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_UserDefHandle,
        L4_ThreadId_t *old_pager);
```

© 2009 University of Karlsruhe, System Architecture Group

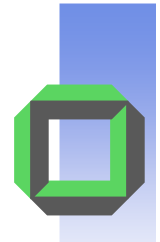


Thread management



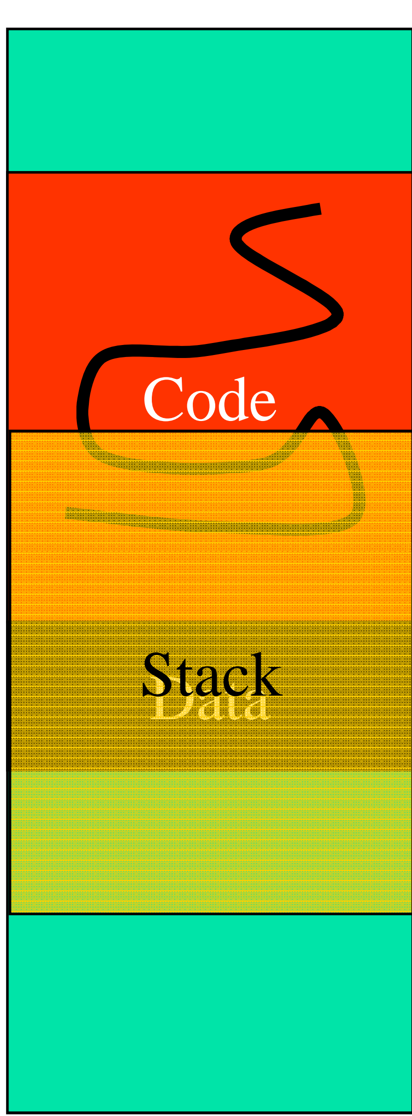
- The microkernel only preserves the user-level IP and SP
  - … and registers if preempted
- Everything else is managed by user-level applications
  This means by you!
  - User stack area
    - Allocation, size, deallocation
  - Thread identifiers
    - Allocation, deallocation
  - Entry point – initial IP/SP values
  - Thread exit

© 2009 University of Karlsruhe, System Architecture Group

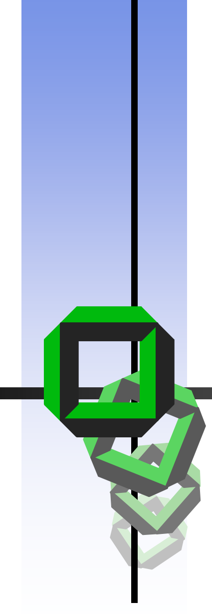


Stack corruption

- Common beginner's problem
- Really weird failure scenarios
  - First printf works, second fails
  - Pointer messed up after calling foo()
  - Random exception when returning
  - ...
- Hard to diagnose/debug
  - Corruption of completely unrelated code and/or data
  - Adding debug code makes problem go away
  - Works fine when single-stepping

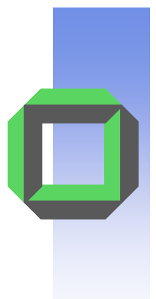


© 2009 University of Karlsruhe, System Architecture Group



Communication

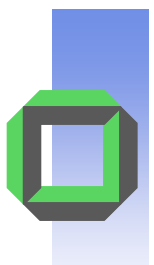
(Ignoring Address Spaces)



IPC Registers

- Message Registers
  - 64 “registers”
  - Form a message
  - Used to transfer typed items and untyped words
    - Typed items
      - StringItem
      - MapItem
      - GrantItem
- Buffer Registers
  - 34 “registers”
  - Specify where typed items are received
    - Typed items
      - StringItem
      - MapItem
      - GrantItem

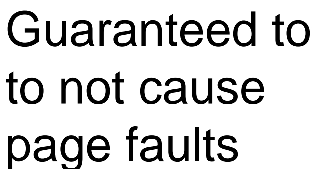
  if any are permitted to be in the message



Message Register Only IPC

Thread A
MR63
…
MR13
MR12
MR11
MR10
MR9
MR8
MR7
MR6
MR5
MR4
MR3
MR2
MR1
MR0

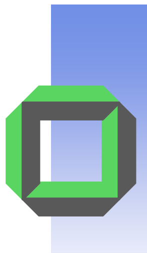
Message transferred from one thread's MRs to the other thread's MRs



Guaranteed to to not cause page faults

Thread B
MR63
…
MR13
MR12
MR11
MR10
MR9
MR8
MR7
MR6
MR5
MR4
MR3
MR2
MR1
MR0

© 2009 University of Karlsruhe, System Architecture Group



Overview of IPC operations

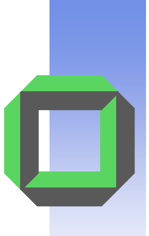
- L4_Ipc system call performs all IPC operations
- Arguments determine actual operation
- Helper functions for frequent operations (see <l4/ipc.h>)
  - L4_Send
    - Send a message to a thread (blocking)
  - L4_Receive
    - Receive a message from a specified thread
  - L4_Wait
    - Receive a message from any sender
  - L4_ReplyWait
    - Send a response to a thread and wait for the next message
  - L4_Call
    - Send a message to a particular thread and wait for it to respond (usual RPC operation)

© 2009 University of Karlsruhe, System Architecture Group

MR_0

- Message content specified by sender's MR_0
  - *u* - number of untyped words
  - *t* - number of words holding typed items
  - *label* - free for the sender to use as part of the message (usually a "label" or "tag")
  - *flags* - specifies option for the IPC operation
    - E.g., propagated message
    - Not used for SDI project (set = 0)

© 2009 University of Karlsruhe, System Architecture Group



Example: Sending 4 untyped words

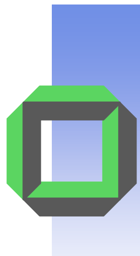
```
L4_Msg_t msg;
L4_MsgTag_t tag;

L4_MsgClear(&msg);
L4_MsgAppendWord(&msg, word1);
L4_MsgAppendWord(&msg, word2);
L4_MsgAppendWord(&msg, word3);
L4_MsgAppendWord(&msg, word4);
L4_MsgLoad(&msg);

tag = L4_Send(tid);
```

MR5			
word 4			
word 3			
word 2			
word 1			
label	*0*	*0*	*4*

© 2009 University of Karlsruhe, System Architecture Group

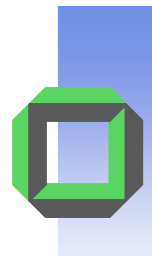


IPC result MR_0

- Message result in receiver's MR_0
- MsgTag [MR_0]
  - *u* - untyped words received ($u = 0$, send only IPC)
  - *t* - typed words received ($t = 0$, send only IPC)
  - Flags EXrp
    - E: error occurred (send or receive), see ErrorCode TCR for details
    - X: received cross processor IPC (ignore)
    - r: received redirected IPC (ignore)
    - p: received propagated IPC (ignore)

$label_{(48)}$	$flags_{(4)}$	$t_{(6)}$	$u_{(6)}$

© 2009 University of Karlsruhe, System Architecture Group



The StringItem Type

- Specifies base address and length
  - Uninterpreted block of bytes
- Used to send a message in place
  - Avoid marshalling costs
- Example sends a single simple string + two untyped words

Note: The typed items always follow the untyped words

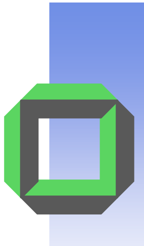
C: specifies whether more typed items follow (redundant with t in MR_0)

:

MR5			
string ptr			
String size	***0***	***0***	***000C***
word 2			
word 1			
label	***0***	***2***	***2***

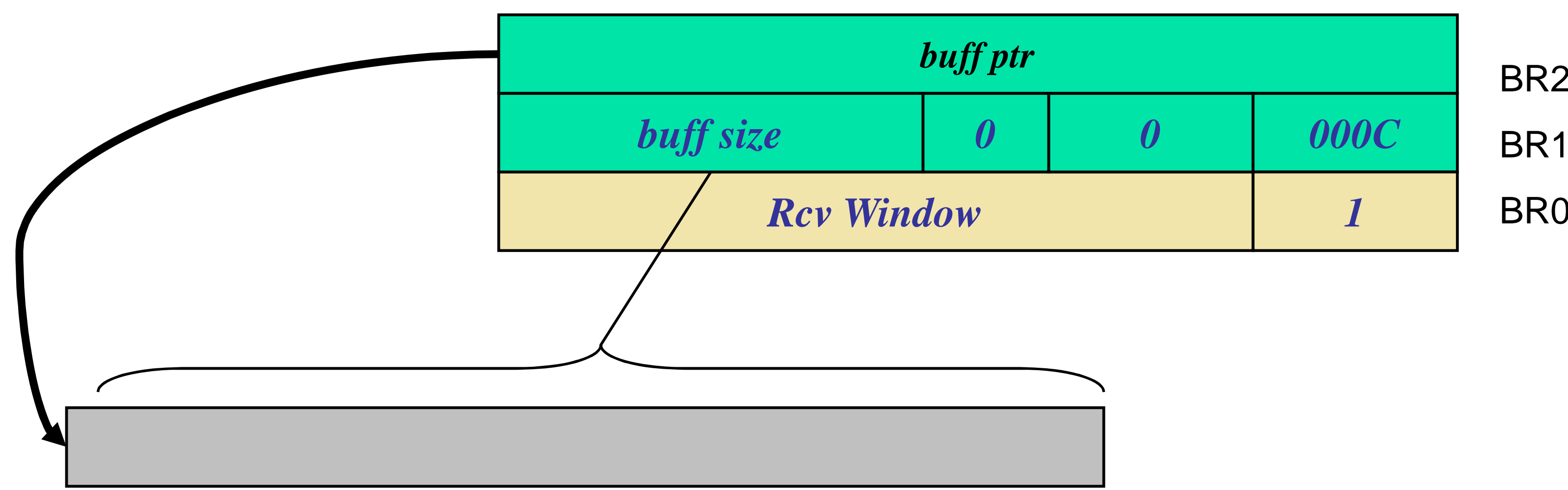
In-memory message

© 2009 University of Karlsruhe, System Architecture Group

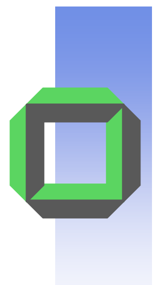


Receiving Strings

- Buffer Registers used to specify area and size of memory region to receive strings
- Simple example
  - A single receive buffer



© 2009 University of Karlsruhe, System Architecture Group



IPC Timeouts

- Used to bound the duration of IPC

- Two *timeout* types

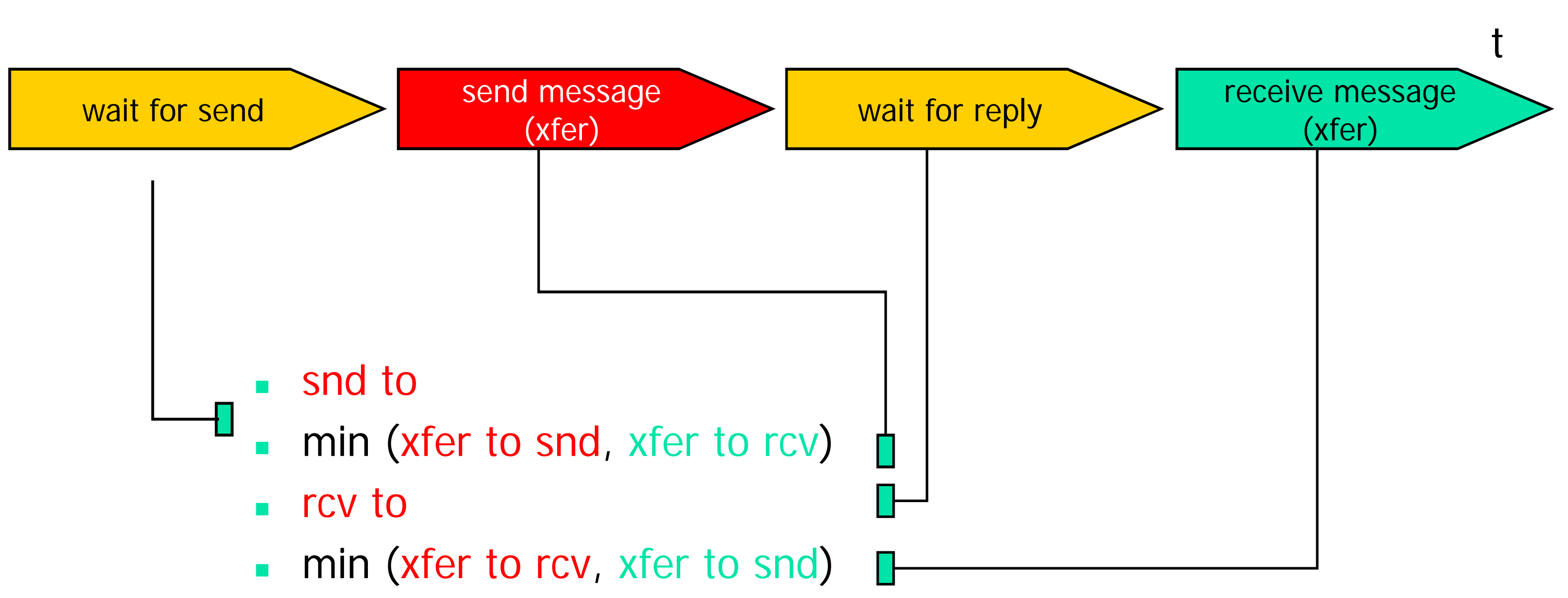
  - Receive/Send Timeouts
    - Used to control how long the IPC syscall will block prior to
      - The send phase beginning (*SndTimeout*)
      - The receive phase beginning (*RcvTimeout*)

  - XferTimeouts (Snd/Rcv)
    - Used to limit how long the IPC transfer takes
      - Only used for StringItems (Why?)
      - Limit time waiting for sender/receiver pagefaults on memory

© 2009 University of Karlsruhe, System Architecture Group

Timeouts

- snd timeout, rcv timeout, xfer timeout snd, xfer timeout rcv

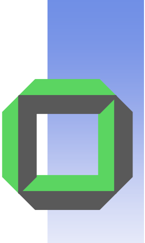


© 2009 University of Karlsruhe, System Architecture Group

Timeouts

- Specifying timeouts
  - Mantissa/exponent representation
  - Relative timeout values
    - 0 — $0_{(16)}$
    - infinite — 0 | $1_{(5)}$ | $0_{(10)}$
    - 1μs … 610 h (log) — 0 | $e_{(5)}$ | $m_{(10)}$ — $2^e m\ \mu s$

© 2009 University of Karlsruhe, System Architecture Group



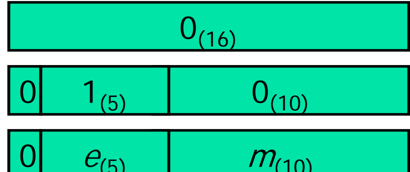
Timeout Value Range

e	m = 1	m = 1023	e	m = 1	m = 1023
0	1.00E-06	1.02E-03	16	6.55E-02	6.70E+01
1	2.00E-06	2.05E-03	17	1.31E-01	1.34E+02
2	4.00E-06	4.09E-03	18	2.62E-01	2.68E+02
3	8.00E-06	8.18E-03	19	5.24E-01	5.36E+02
4	1.60E-05	1.64E-02	20	1.05E+00	1.07E+03
5	3.20E-05	3.27E-02	21	2.10E+00	2.15E+03
6	6.40E-05	6.55E-02	22	4.19E+00	4.29E+03
7	1.28E-04	1.31E-01	23	8.39E+00	8.58E+03
8	2.56E-04	2.62E-01	24	1.68E+01	1.72E+04
9	5.12E-04	5.24E-01	25	3.36E+01	3.43E+04
10	1.02E-03	1.05E+00	26	6.71E+01	6.87E+04
11	2.05E-03	2.10E+00	27	1.34E+02	1.37E+05
12	4.10E-03	4.19E+00	28	2.68E+02	2.75E+05
13	8.19E-03	8.38E+00	29	5.37E+02	5.49E+05
14	1.64E-02	1.68E+01	30	1.07E+03	1.10E+06
15	3.28E-02	3.35E+01	31	2.15E+03	2.20E+06

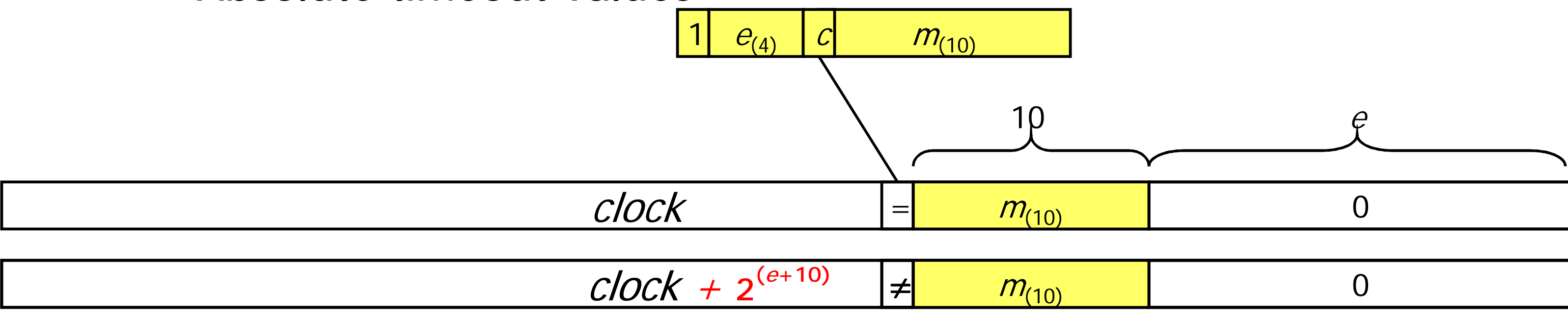
© 2009 University of Karlsruhe, System Architecture Group

Timeouts

- Specifying timeouts
  - Mantissa/exponent representation
  - Relative timeout values
    - 0
    - infinite
    - $1\mu s$ … 610 h (log)
  - Absolute timeout values



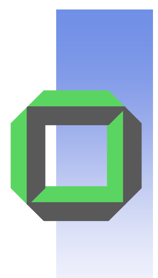
$2^e m\ \mu s$



© 2009 University of Karlsruhe, System Architecture Group

IPC system call

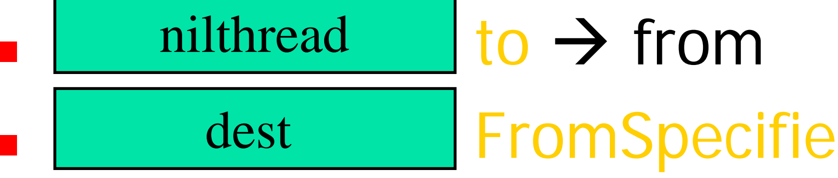
- to → from
- FromSpecifier
- Timeouts
- MR_0



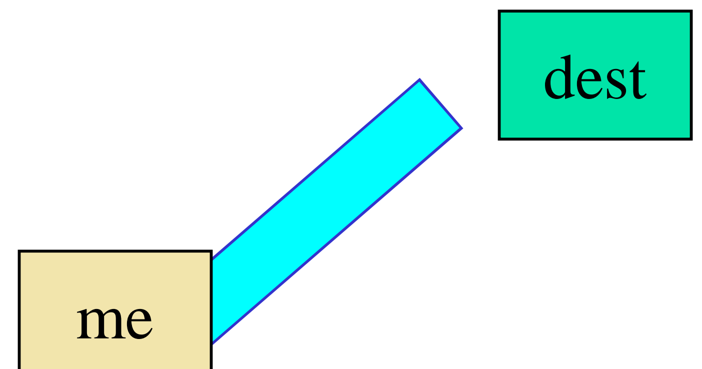
IPC system call

- Receive

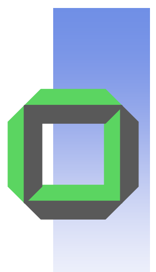
  from *dest*



- to → from
- FromSpecifier
- Timeouts
- MR_0

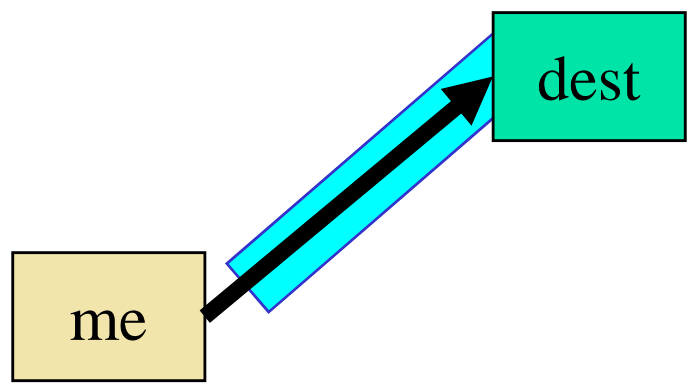


© 2009 University of Karlsruhe, System Architecture Group



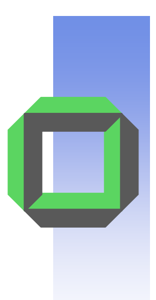
IPC system call

- Send



- dest to → from
- nilthread FromSpecifier
- Timeouts
- MR_0

© 2009 University of Karlsruhe, System Architecture Group

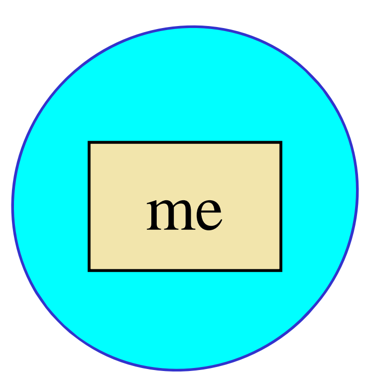


IPC system call

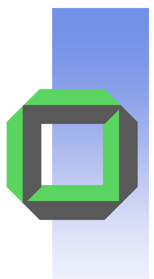
- Wait

  Receive from *anyone*

  - nilthread to → from
  - anythread FromSpecifier
  - Timeouts
  - MR_0

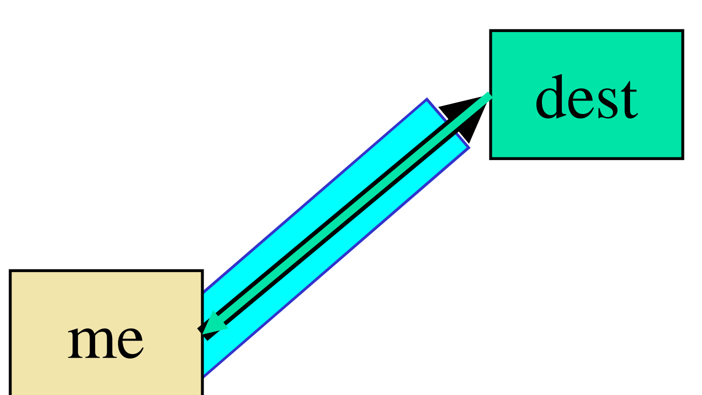


© 2009 University of Karlsruhe, System Architecture Group



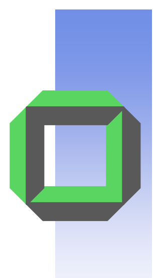
IPC system call

- Call



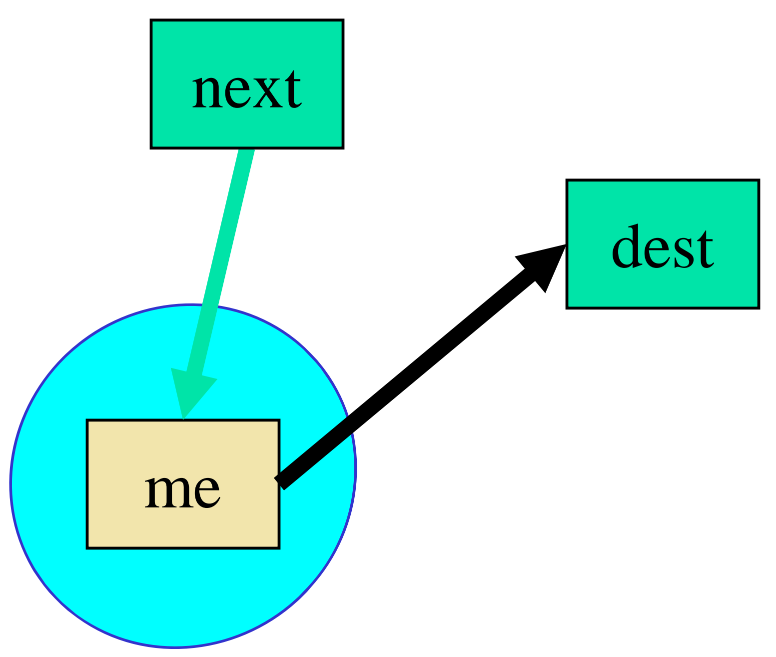
- dest to → from
- dest FromSpecifier
- Timeouts
- MR_0

© 2009 University of Karlsruhe, System Architecture Group



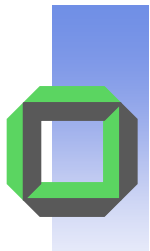
IPC system call

- ReplyWait



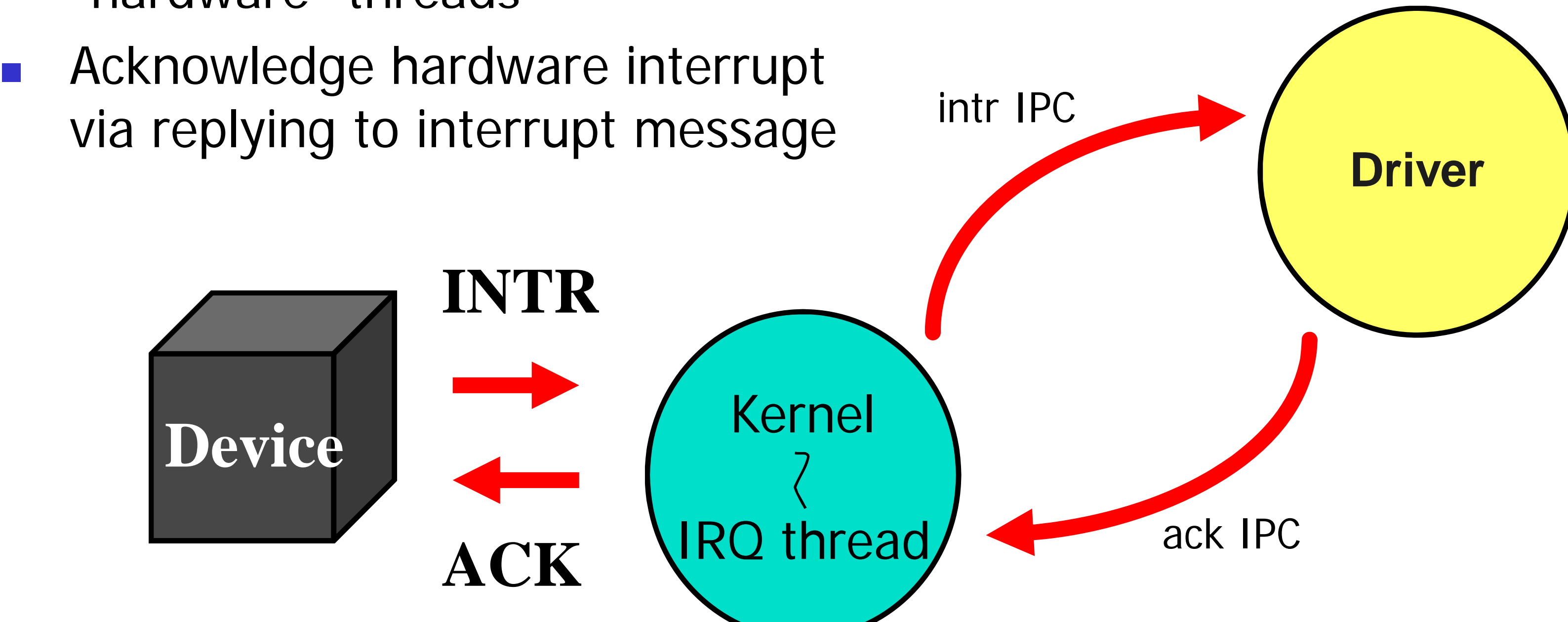
- dest to → from
- anythread FromSpecifier
- Timeouts
- MR_0

© 2009 University of Karlsruhe, System Architecture Group



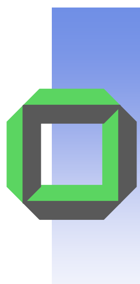
Interrupts

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



The interrupt message is sent to the hardware thread's "pager"

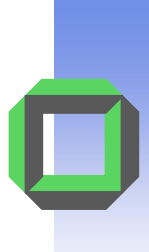
© 2009 University of Karlsruhe, System Architecture Group



Interrupt Association

- Association is done via the privileged thread (root task) using **`ThreadControl.`**
- To associate a thread to an interrupt
  - Set the pager of the hardware thread ID to the thread ID of the interrupt handler
- To disassociate the thread from an interrupt
  - Set the pager of the hardware thread ID to the hardware thread ID itself

© 2009 University of Karlsruhe, System Architecture Group



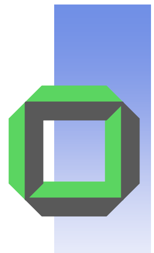
Sample Code

```
L4_ThreadId_t tid;
int res;

tid.global.X.thread_no = irq;
tid.global.X.version = 1;

res = L4_ThreadControl(tid,         /* irq thread id */
                       tid,
                       L4_nilthread,
                       driver_tid, /* pager, the
                                      thread we
                                      want the irq
                                      to be associated
                                      with */
                       (void*) -1);
if (res != 1) {
   printf("BADNESS ON THREAD CONTROL\n");
}
```

© 2009 University of Karlsruhe, System Architecture Group



Microkernel System Calls

KernelInterface

IPC

Unmap

ExchangeRegisters

ThreadSwitch

Schedule

SystemClock

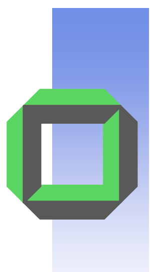
ThreadControl

SpaceControl

ProcessorControl

MemoryControl

© 2009 University of Karlsruhe, System Architecture Group

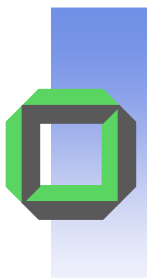


ThreadSwitch

- Yields the current thread's remaining time slice, or donates remaining time slice to another thread
  - You should not need to do this, but…
    - Can use to reduce impact of busy-wait
    - Ensure progress of resource (lock) holder

- **`L4_ThreadSwitch (thread);`**
  - Thread = nilthread: Yield the processor
  - Thread = threadID: Donate time slice
  - See <l4/schedule.h> for derived functions

© 2009 University of Karlsruhe, System Architecture Group

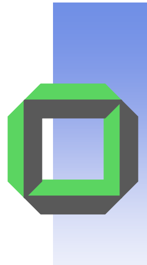


Schedule

- L4 implements a mostly multi-level round robin scheduler
  - Static priorities
  - Time slice donation on IPC

- **`Schedule`** is used
  - To change scheduling parameters of threads
    - Priority
    - Time slice
    - Total quantum (don't use, set to infinity)
  - For controlling preemption parameters
    - Not implemented
  - Set the processor the thread should run on
    - Not needed, you have only one

© 2009 University of Karlsruhe, System Architecture Group



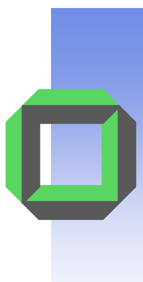
Schedule

- Only a thread's scheduler can invoke the system call
- The scheduler is set using thread control
  - Typically the root task will remain the scheduler

```
L4_Schedule (L4_ThreadId_t dest,
             L4_Word_t TimeControl,
             L4_Word_t ProcessorControl,
             L4_Word_t prio,
             L4_Word_t PreemptionControl,
             L4_Word_t * old_TimeControl)
```

- Derived functions in <l4/schedule.h>

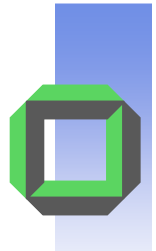
© 2009 University of Karlsruhe, System Architecture Group



SystemClock

- Returns the current system clock
  - 64-bit number that counts µ-seconds since boot-up
- Not always a real system call (no kernel entry)
  - i.e., may use user-accessible processor cycle counter

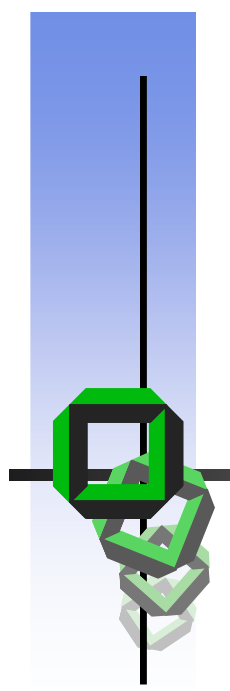
© 2009 University of Karlsruhe, System Architecture Group



Microkernel System Calls

KernelInterface
IPC
Unmap
ExchangeRegisters
ThreadSwitch
Schedule
SystemClock
ThreadControl
SpaceControl
ProcessorControl
MemoryControl

© 2009 University of Karlsruhe, System Architecture Group

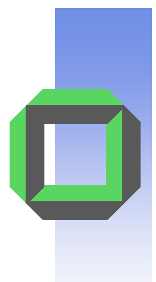


System Design and Implementation

L4 API Crash Course

Part II

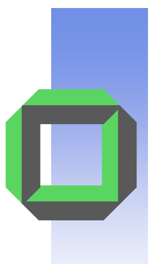
... next week



Next week

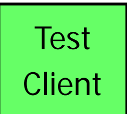
- Change your group password
- Get your build environment going
  - See Wiki (http://i30www.ira.uka.de/teaching/courses/sdi)
  - Room 149 is accessible till 6pm (often longer)
  - Wiki User: student Password: sdi2009

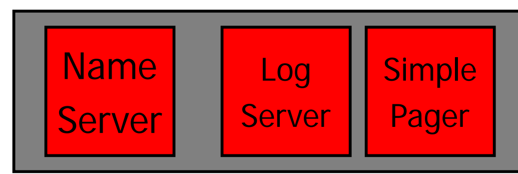
© 2009 University of Karlsruhe, System Architecture Group



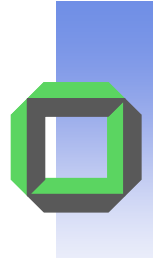
Example Code

- Binaries
  - KickStart – 3rd stage loader
  - Kernel
  - Sigma0
- Root Task
  - Name server
  - Log Server
  - Pager
  - Starts the test task
- Test Task
  - Uses name server to locate log server
  - Prints message to the log



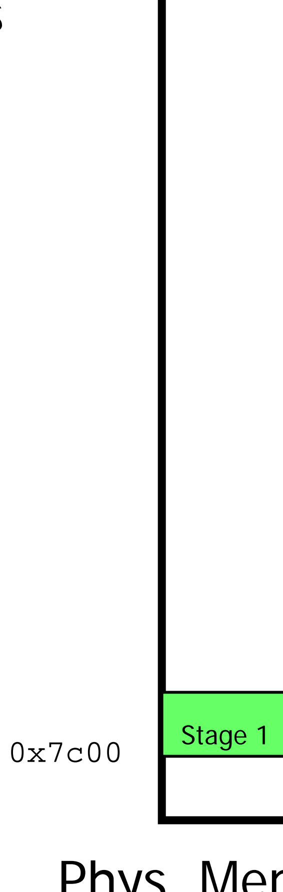


© 2009 University of Karlsruhe, System Architecture Group

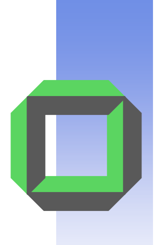


The Boot Sequence

- BIOS loads the boot block – GRUB stage1
  - Stage1 is a simple loader that fits into 512 bytes
  - Responsible for loading stage2

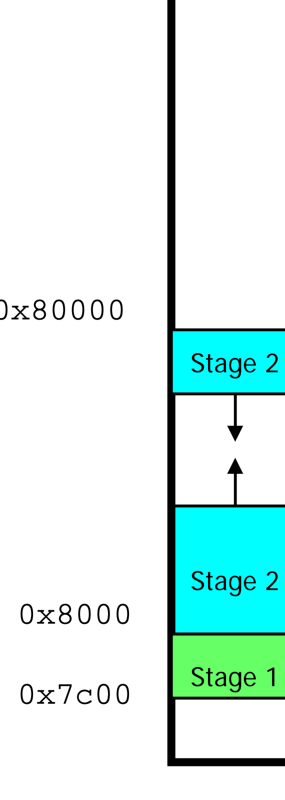


© 2009 University of Karlsruhe, System Architecture Group



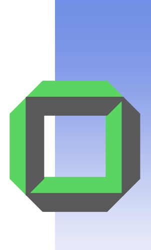
The Boot Sequence

- Stage 2 is loaded by stage 1
  - Stage 2 is a more complex loader that
    - Speaks various file system formats
    - Supports loading via network
    - Supports a menu providing a choice of load configurations – menu.lst
    - Approx 60Kb - 80Kb in size + stack and heap
    - Supports ELF loading



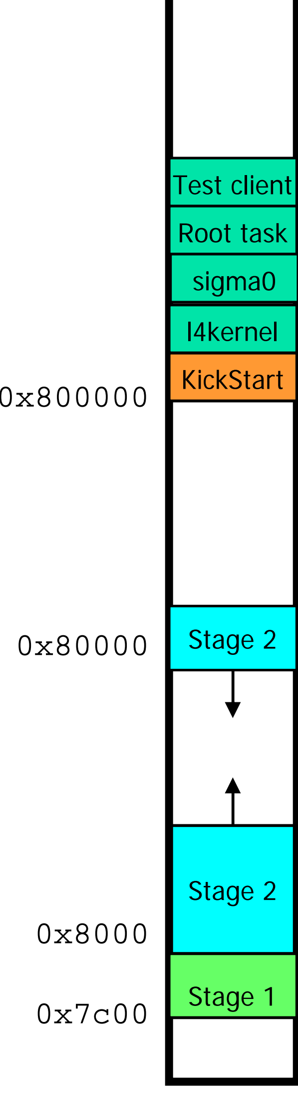
Phys. Mem

© 2009 University of Karlsruhe, System Architecture Group

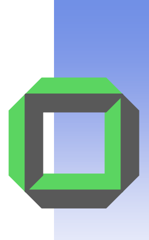


The Boot Sequence

- KickStart is ELF loaded at its linked address (8MB)
- GRUB thinks this is the kernel
- Modules are appended after the kernel
  - Modules are loaded beginning on page boundaries
- A multiboot header is generated based on the modules loaded
- KickStart is started and passed a pointer the multiboot header

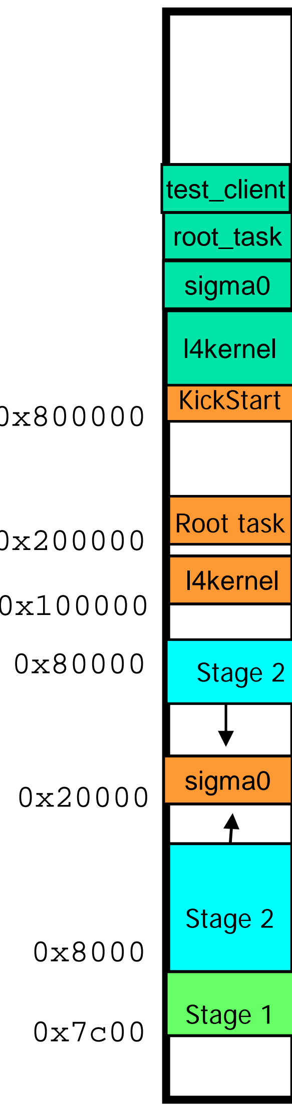


© 2009 University of Karlsruhe, System Architecture Group



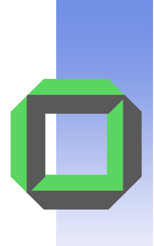
The Boot Sequence

- KickStart copies the multiboot info
- It ELF loads
  - The L4kernel (at 0x100000),
  - Sigma0 (at 0x20000)
  - Root task (at 0x200000).
- It configures the kernel
- It finally starts the L4 kernel



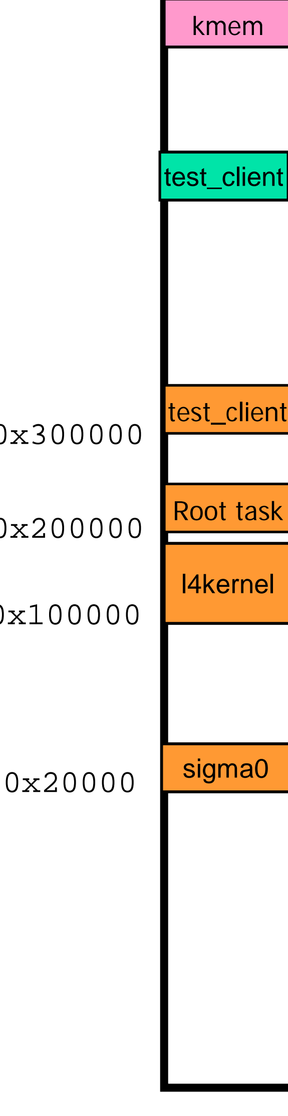
Phys. Mem

© 2009 University of Karlsruhe, System Architecture Group



The Boot Sequence

- L4 starts.
- The kernel grabs some upper memory for page tables etc.
- L4 starts sigma0 and then starts the root task.
- Root task loads and starts test_client



Phys. Mem

© 2009 University of Karlsruhe, System Architecture Group