seL4 API and Libraries

Details and Hands-on Exercises

Ihor Kuz, Stephen Sherratt, Kofi Doku Atuah

August 2016

www.csiro.au

CSIRO

Agenda and Exercises

Agenda

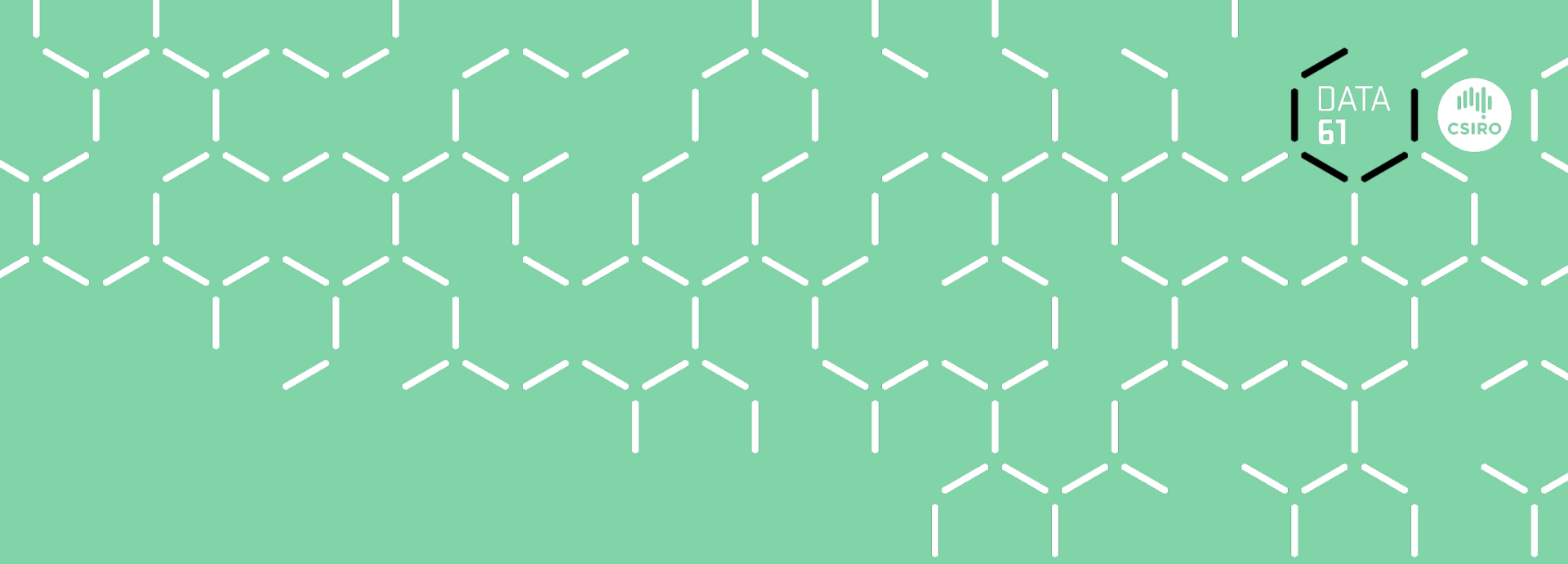
- Building and running
- Finding code and debugging
- System startup
  - Root task and bootinfo
- Untypeds and Allocators
- Threads and IPC
- CSpaces, VSpaces
- Processes
  - Vspace library
  - Sel4utils library

Exercises

- Hello-1:
  - simple hello world
- Hello-2:
  - start a new thread
- Hello-3:
  - IPC between threads
- Hello-4:
  - start a new process
  - IPC between processes
- Platform: ia32 (x86)
  - Bonus: make it work for ARM

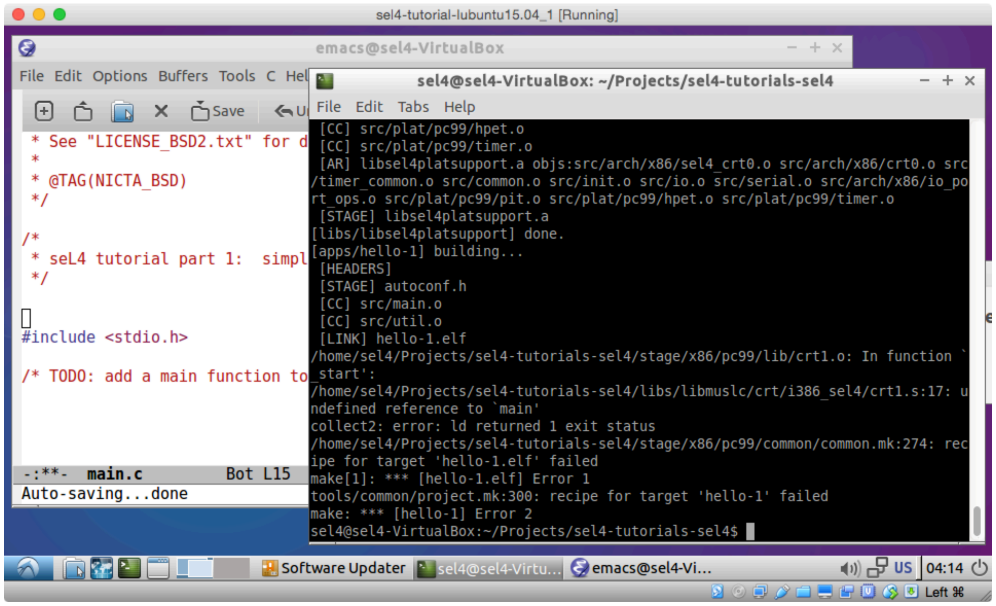
Prerequisites

- Knowledge prereqs (what we expect)
  - Not required:
    - Previous seL4 experience
  - Required:
    - Be a quick learner!
    - C programming
    - Unix environment (editing files, running build tools, etc.)
- Technical prereqs (what you should have working already)
  - Linux (will work on BSD, MacOSX with some extra work, cygwin maybe?)
  - C compilers and build tools (gcc)
  - Python and packages
  - Haskell, cabal, and packages
  - Git and repo
  - qemu



Building and running a basic seL4 program - Hello World

VirtualBox VM



- VirtualBox VM contents:
  - Linux: Lubuntu 16.04
    - login: sel4:sel4
  - Prerequisites installed
  - Tutorial code
    - In ~/Projects/
  - Tutorial slides & seL4 docs

Install and run

- Get Virtualbox (newest: 5.1.2)
- Install it
  - Also get and install the Extension Pack
- Get seL4 tutorial “appliance”
- Import it, Run it

Getting the code

- Using repo (and git)

```
mkdir sel4-tutorials
cd sel4-tutorials
repo init –u http://github.com/sel4proj/sel4-tutorials-manifest
       -m sel4-tutorials.xml
repo sync
```

- ***Root directory***
  - The directory where you did `repo init`
- Exercises
  - In apps/hello-*
  - Contain program skeletons
    - Interesting bits replaced with “TODO” comments
    - Includes lots of hints
  - Goal is to add code for the TODOs to create complete programs

Working Directory

- What's there?
  - Config and build files: Kconfig, Kbuild, Makefile
  - kernel/ - from https://github.com/seL4/seL4
  - libs/ - from https://github.com/seL4/{sel4_libs, util_libs}
  - projects/
    - sel4-tutorials/ - from https://github.com/seL4proj/sel4-tutorials
      - docs/, run-arm.sh, run-ia32.sh
  - apps/
    - hello-*/
      - src/*.c, Kbuild, Kconfig, Makefile
  - configs/
    - *_defconfig
  - tools/
  - - build system (.mk files) and build tools

Documentation

- seL4 manual
  - http://sel4.systems/Info/Docs/seL4-manual-latest.pdf
- CAmkES manual
  - https://github.com/seL4/camkes-tool/blob/master/docs/index.md
- Code
  - seL4 API
    - https://github.com/seL4/seL4/tree/master/libsel4
    - kernel/libsel4
  - Libraries
    - https://github.com/seL4/{sel4_libs, util_libs}
    - libs/*
- These slides
- seL4 wiki
  - https://wiki.sel4.systems/Getting%20started#Start_with_the_SEL4_tutorials

Configuration

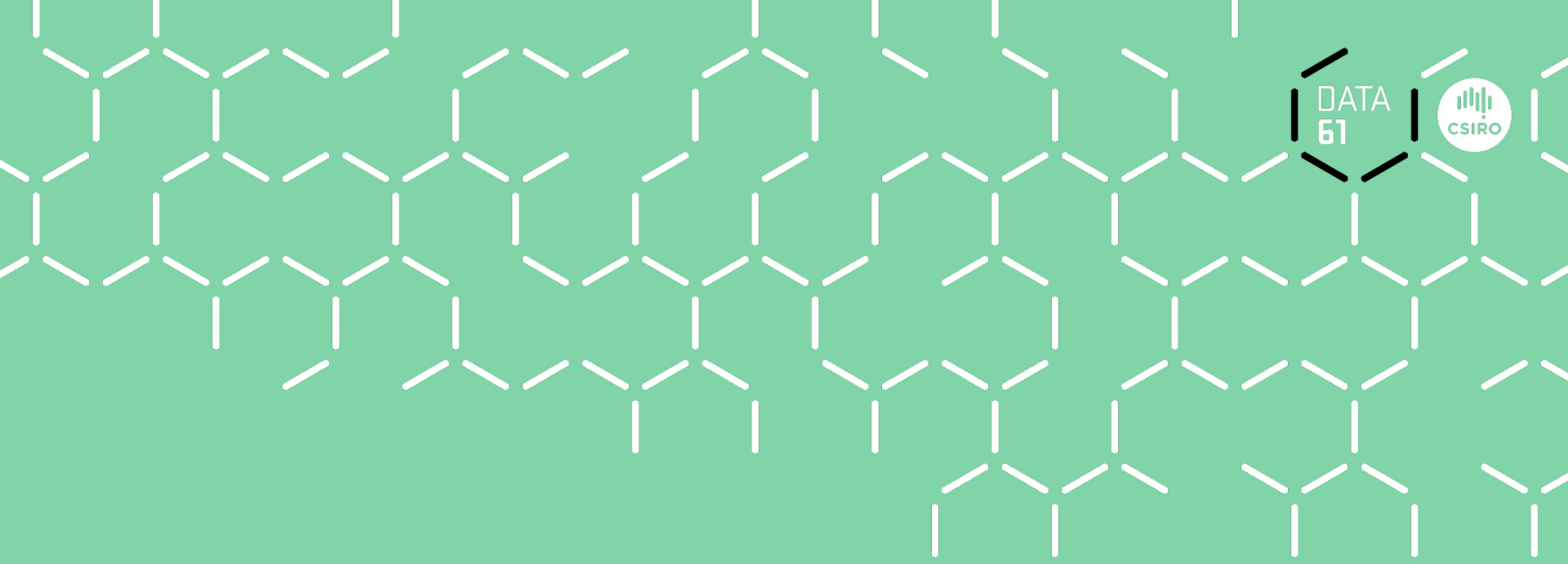
- Based on the Linux Kernel Build System: Kbuild
  - `Kconfig`: define config symbols and their attributes
  - `Kbuild`: defines dependencies between modules
  - `Makefile`: make rules (uses many .mk files from tools/common)
  - `.config`: stores each config symbol's selected value
- make menuconfig
  - Curses-based interface, set various config options
  - Writes a .config file
- configs/
  - Default configurations (copies of .config files for different configurations)
  - E.g.: ia32_hello-0_defconfig
- make …defconfig
  - E.g. make ia32_hello-0_defconfig
  - Copies configs/ia32_hello-0_defconfig to .config

Building and Running

- `make`
  - Builds required modules based on .config
  - Compilation occurs outside of source directory
    - `build/`: generated files and results of compilation
    - `stage/`: intermediate results, copies of header files, etc.
  - `images/`
    - Loadable system image files. Example:
      - Kernel: `kernel-ia32-pc99`
      - User: `hello-0-image-ia32-pc99`
- Run using qemu: system emulator
  - `qemu-system-i386 -nographic -m 512 -kernel <kernel> -initrd <user>`
- `make clean, make mrproper`
- Convenience scripts
  - `run-ia32.sh, run-arm.sh`

Hands-on: hello-1

- Goal: Get and build a simple seL4 system
- Get code (if you haven't already)
  - Hint: `mkdir …; cd …; repo init …; repo sync`
- `make ia32_hello-1_defconfig`
  - `make menuconfig`
- `make`
  - Compilation failure!
  - Find and fix TODO 1
- Run:
  - `qemu-system-i386 -nographic -m 512 -kernel images/kernel-ia32-pc99 -initrd images/hello-1-image-ia32-pc99`
  - Quit qemu with: C-A x
- Where's the solution?
  - `projects/sel4-tutorials/solutions/hello-1`



Finding code and Debugging

Looking up code

- Github
  - http://github.com/sel4
  - Lookup appropriate repo (sel4, sel4_libs, util_libs, etc.)
  - Search (github search isn't great)
- Cscope
  - Search C code
  - Run directly: `cscope -R`
    - <Tab> between fields
    - Choose file to view
    - Ctrl-D to exit
- Run cscope in browser
  - Vim:
    - http://cscope.sourceforge.net/cscope_vim_tutorial.html
    - :cscope f g <function>, C-t
    - C-\ g, C-<space> g
  - Emacs: xcscope.el
    - https://github.com/dkogan/xcscope.el
  - C-c s s

Hands-on: looking up code

- Find these functions:
  - simple_default_init_bootinfo()
  - allocman_make_vka()
  - vka_alloc_tcb()
  - seL4_TCB_Configure()
- Use:
  - Github
  - Cscope
- What about libsel4?
  - It is generated at build time!
  - libsel4/include/interfaces/sel4.xml
  - build/…/libsel4/include/interfaces/sel4_client.h

Build detail

- Build detail
  - Increase detail with V=
  - Example:
    - make V=0
    - make V=1
    - make V=2
    - make V=3
  - Useful to find out why a build step fails
    - What is being run
    - What are the arguments (e.g., include paths, or library paths)

Debugging

- VM fault

```
Caught cap fault in send phase at address 0x0
while trying to handle:
vm fault on data at address 0x0 with status 0x6
in thread 0xffaf9900 "rootserver" at address 0x80480db
```

- use objdump
  - `objdump -dS build/x86/pc99/hello-1/hello-1.bin | less`
  - look for instruction at address: 80480db

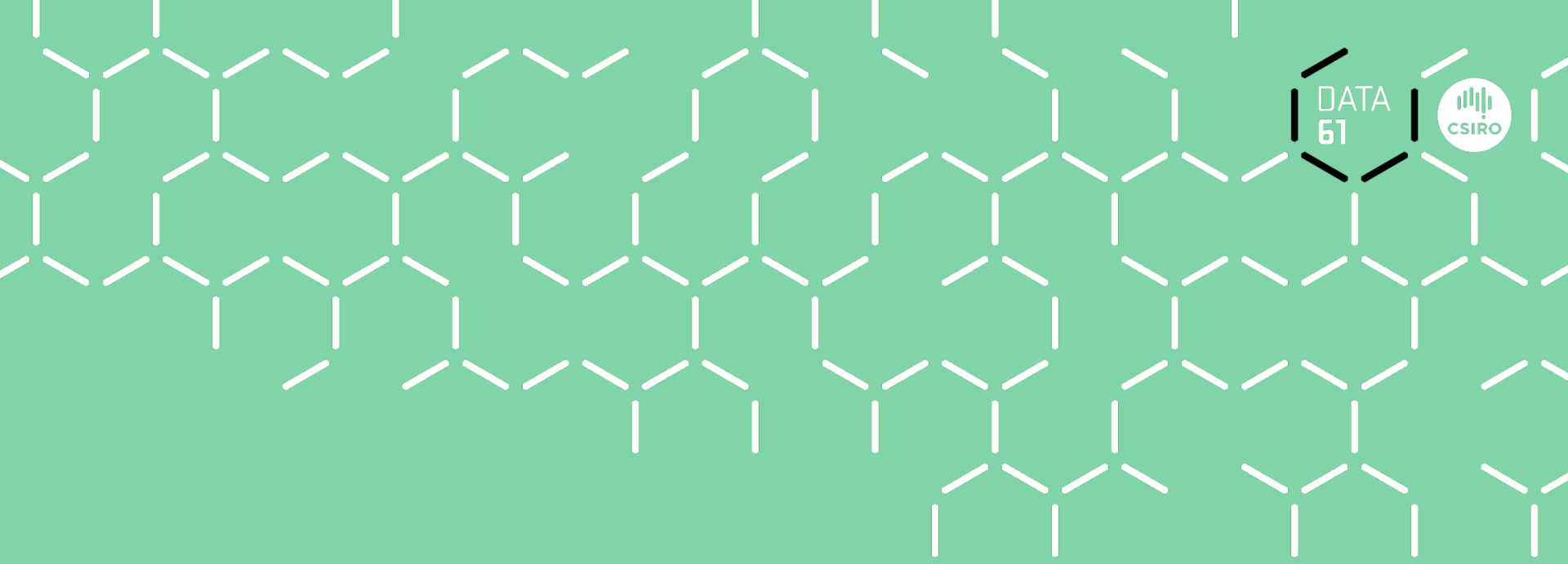
```
printf("hello world\n");
 80480d1:       68 40 c8 04 08    push   $0x804c840
 80480d6:       e8 c8 02 00 00    call   80483a3 <puts>
    *(char*)0x0 = 'a';
 80480db:       c6 05 00 00 00 00 00  movb $0x0,0x0
 80480e2:       0f 0b                 ud2
```

Hands-on: hello-1 with errors

- Goal: Introduce an error into hello-1
- Add a line in main():
  - `*(char*)0x0 = 'X';`
- Build and run it -> VM fault
- Find it
  - Use objdump to find it
- Fix it ☺

More debugging:

- https://wiki.sel4.systems/Debugging%20guide



seL4 System Startup

- Root task and Bootinfo
- libsimple

seL4 system startup ®

- Image
  - Kernel image
  - User-space image
    - Root task & Cpio file containing elf files
- Boot loader
  - Loads kernel into memory
  - Loads user-space image into memory
  - Starts kernel running
- Kernel startup
  - Kernel creates object (untypeds, frames for device memory)
  - Kernel creates root task objects
  - Loads and runs root task
- Root Task
  - Responsible for setting up the rest of the system

BootInfo: Start-up Information ®

- Kernel creates:
  - root task CSpace, root task VSpace, Root task TCB
  - frames for device memory
  - untyped caps for RAM memory
- All startup objects are available to root task
  - kernel places caps to these objects in root task CSpace
  - kernel needs to tell root task
    - what caps it has
    - what the objects are
- Bootinfo
  - info about all the initial objects and the caps to them

Root task CSpace and Bootinfo ®

CSpace

Bootinfo

Name	Data	Description
empty	Start slot, end slot	Empty CSpace slots
userImageFrames	Start slot, end slot	Slots with root task image's (code, data) frame caps
userImagePaging	Start slot, end slot	Slots with PD and PTs for root task VSpace
untyped	Start slot, end slot	Slots with untyped object caps (sorted by size)
untypedPaddrList	Array of addresses	Physical address for each untyped object
deviceRegions	Array of {paddr, size, start slot, end slot}	Information about all device memory

Initial caps

- Some Initial Caps
  - seL4_CapNull = 0, /* null cap */
  - seL4_CapInitThreadTCB = 1, /* initial thread's TCB cap */
  - seL4_CapInitThreadCNode = 2, /* root CNode cap */
  - seL4_CapInitThreadVSpace = 3, /* VSpace cap */
  - seL4_CapBootInfoFrame = 9, /* bootinfo frame cap */
  - seL4_CapInitThreadIPCBuffer = 10, /* initial thread's IPC buffer frame cap */
- Some bootinfo fields
- seL4_SlotRegion untyped; /* untyped-object caps (untyped caps) */
- seL4_Word untypedPaddrList[…]; /* physical address of each untyped cap */
- seL4_Uint8 untypedSizeBitsList[…]; /* size (2^n) bytes of each untyped cap */
- seL4_Word numDeviceRegions; /* number of device regions */
- seL4_DeviceRegion deviceRegions[…]; /* device regions */

Intro to seL4 Libraries ®

- Goal:
  - Make seL4 programming less “user-unfriendly”
  - Do a bunch of the hard things for you
- Interfaces vs Implementations
  - *Interface*
    - key datastructs
    - function definitions
    - generic code to facilitate use of interface
  - *Implementation*
    - adds implementation-specific parts to datastructs
    - implements interface functions

Key interfaces and libraries ®

- Key Interfaces
  - **simple**: access to initial caps
  - **vka**: virtual kernel allocator
  - **vspace**: VSpace management
- Key Libraries
  - **libseL4**: seL4 kernel API
  - **allocman** (vka): allocator manager
  - **sel4utils** (vspace, io operations): higher level concepts
- Other Libraries
  - **muslc**: C library
  - **platsupport**, **sel4platsupport**: device access
  - **utils**, **debug**, **benchmark**: other useful functionality

Dependencies

- simple:
  - libsel4
- vka:
  - libsel4, utils
- allocman:
  - vka, libsel4, sel4utils, vspace, utils
- vspace:
  - vka, libsel4, utils
- sel4utils:
  - simple, vka, vspace, platsupport, sel4utils, utils, elf, cpio
- platsupport:
  - utils
- sel4platsupport:
  - simple, vka, vspace, platsupport, sel4utils, utils

Library: Simple ®

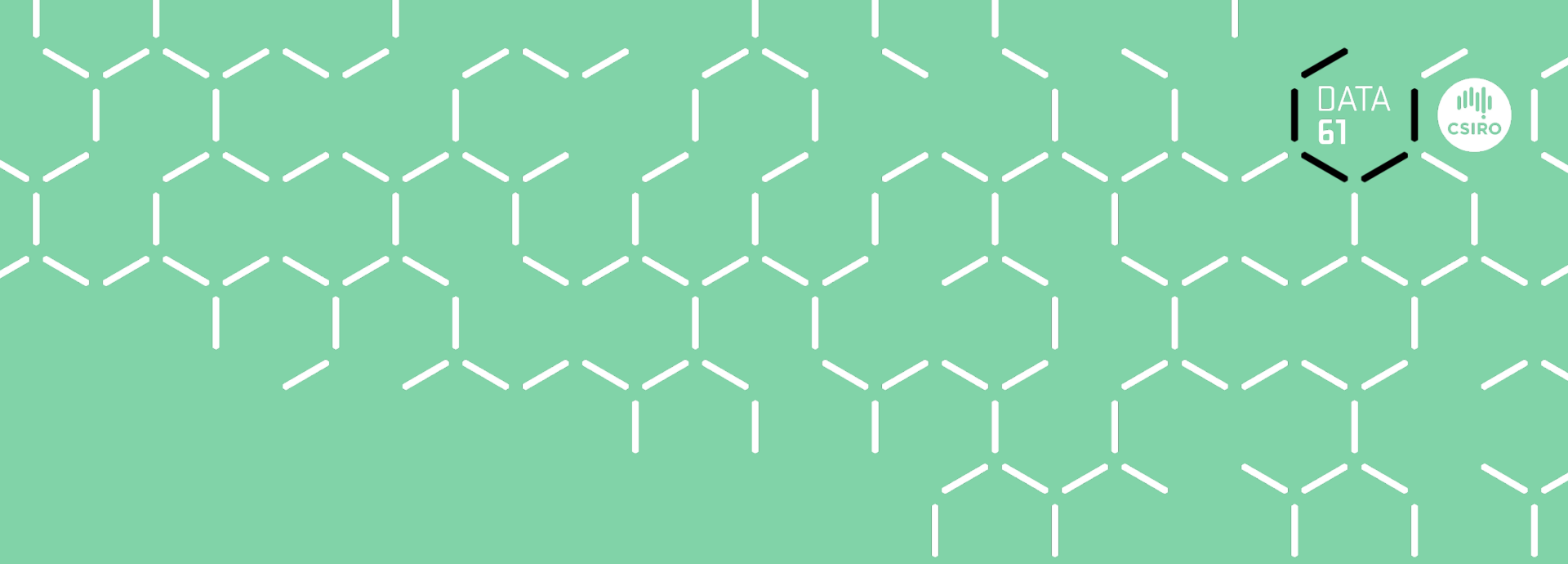
- Easy way to access initial caps
  - Includes: untypeds, device memory, initial CSpace, initial VSpace
- Abstracts over spec of initial caps
  - root task: uses bootinfo
  - user-level task: can use bootinfo or some other format
- Key concepts
  - location of resources, caps to resources
  - acquiring resource without cap
- Interfaces defined
  - `Simple`
- Implemented by
  - simple-default, simple-stable, simple-camkes

Simple: API

- Files
  - libsel4simple, #include <simple/simple.h>
    - `libs/libsel4simple/include/simple/`
  - libse4simple-default, #include <simple-default/simple-default.h>
    - `libs/libsel4simple-default/include/simple-default/`
- Datastructs
  - `simple_t`
- Functions
  - `simple_default_init_bootinfo` (simple-default.h)
  - `simple_print`
  - `simple_get_*`: pd, tcb, cnode, node_size
  - `simple_get_nth_untyped`
  - `simple_get_frame_*`: cap, info, vaddr

Hands-on: hello-2 (part 1)

- Goal: Initialise a simple and look at bootinfo
- `make ia32_hello-2_defconfig`
- Edit `apps/hello-2/src/main.c`
- Fix TODOs
  - TODO 1: get bootinfo
  - TODO 2: init simple
  - TODO 3: print out bootinfo
- `make`
- run
  - Print out showing what's in bootinfo
  - See if it makes any sense…



seL4 API and libraries

- Untypeds and Allocators
- Starting a new thread

Untyped and retyping ®

- Untyped Memory Object
  - region of (RAM) memory
  - must be retyped to another object to use it
    - results in a nested tree from a root untyped to other objects:
- Retyping
  - kernel uses part of untyped's memory region to store a new kernel object
    - can only create an object if you have a cap to a big enough untyped object
  - retype provides user with cap to the new object
- seL4_Untyped_Retype
  - `seL4_Untyped_Retype(seL4_Untyped` **`service`**`, int` **`type`**`, int` **`size_bits`**`, seL4_CNode root, int node_index, int node_depth, int node_offset, int` **`num_objects`**`)`

Allocators ®

- Allocating objects requires
  - Pool of untypeds to retype into the new objects
  - CSpace slots to put the cap to the new object into
  - Memory for bookkeeping structures
  - Bookkeeping:
    - What untypeds are available, what sizes are they, where are their caps?
    - How much of the untyped has been used?
    - What CSpace slots are available?
    - Which objects have been created, and which untypeds where used and in which Cspace slots are their caps stored?
- Allocator
  - Manages the untyped pools, the CSpace slots
  - Takes care of bookkeeping
  - Maybe even allows objects to be freed!

Library: VKA ®

- VKA: Virtual Kernel Allocator
- Interface for allocating kernel objects
  - abstracts away
    - creation of objects through retyping untypeds
    - managing CSpace and book keeping
- Key concepts
  - ***vka***: allocator
  - ***Object****s*: represent kernel objects
  - ***CSpace slots***: slot in local cspace where caps can be found
  - ***cspace path***: fully qualified capability address
  - ***utspace***: pool of untyped memory, used to create objects
- Interfaces defined: `vka`
- Implemented by: allocman

VKA: API

- Files
  - libsel4vka, #include <vka/…> : vka.h, object.h
  - `libs/libsel4vka/include/vka/`
- Datastructs
  - `vka_t`: a VKA interface instance
  - `vka_object_t`: VKA representation of a kernel object
    - Contains: cptr, ut cookie, object type, size
- Functions
  - `vka_alloc_*`: pd, cnode, tcb, endpoint. returns vka_object_t
  - `vka_cspace_alloc`: allocate an empty slot in the CSpace
  - `vka_cspace_free`: doesn't delete object!
  - `vka_utspace_alloc`: given empty slot create an object and put the cap to it in the slot. returns ut cookie
  - `vka_utspace_free`: given ut cookie, free object

Library: Allocman ®

- Allocator Manager
  - implements vka interface
  - framework combining independent CSpace and utspace allocators
  - solves difficult recursion problems in allocation: Black Magic!
- Key concepts
  - resources: allocator needs underlying resources (e.g. untypeds)
  - memory pool: needs an initial pool for internal allocations
- Interfaces implemented
  - `vka`
  - `allocman`: to add resources after initialisation

Allocman: API

- Files:
  - libsel4allocman, #include <allocman/…>: allocman.h, vka.h, bootstrap.h
  - `libs/libsel4allocman/include/allocman`
- Datastructs
  - `allocman_t`
- Functions
  - Bootstrap: create new allocman
    - `bootstrap_use_current_simple`
      - use current CSpace and simple
    - `bootstrap_new_2level_simple`
      - create and switch to a new CSpace
  - `allocman_make_vka`
    - get VKA interface to allocman

Hands-on: hello-2 (part 2)

- Goal: Create an allocator
- Fix TODOs:
  - TODO 4: create an allocator
  - TODO 5: create a vka
- Build and run
  - No new visible output.

seL4 API: Overview ®

- Key Concepts
  - Kernel Object
    - in-kernel datastruct, only directly accessible by kernel
  - Capability
    - reference to a kernel object
    - allows holder to invoke functions on the objects
      - i.e. ask kernel to do something with the object
      - holder: thread invoking the cap
- Low-level interface for key activities
  - create kernel objects (retype untyped)
  - create and manage caps in a CSpace
  - create and manage VSpace
  - create and manage threads
  - communicate between threads (IPC)

seL4 API: key files

- Files:
  - Libsel4, #include <sel4/sel4.h>
    - Includes: types.h, bootinfo.h, arch/syscalls.h, interfaces/sel4_client.h
  - `libs/libsel4/`
    - `include/sel4`
    - `arch_include/x86/sel4/`
  - `build/x86/pc99/libsel4/include/interfaces/`
- Generated from
  - `libs/libsel4/include/interfaces/sel4.xml`
- Note:
  - libsel4 actually lives in the kernel
  - `libs/libsel4` is a symlink to `kernel/libsel4`

seL4 Capability ®

- Kernel-maintained
  - user-level cannot directly access or manipulate a capability
  - capability is stored in CSpace
  - pass CSpace address of cap in system calls
- Datatypes
  - `seL4_CPtr`
    - index into current thread's CSpace root (CNode)
    - this can be tricky….

seL4 API: TCB (Thread Control Block)

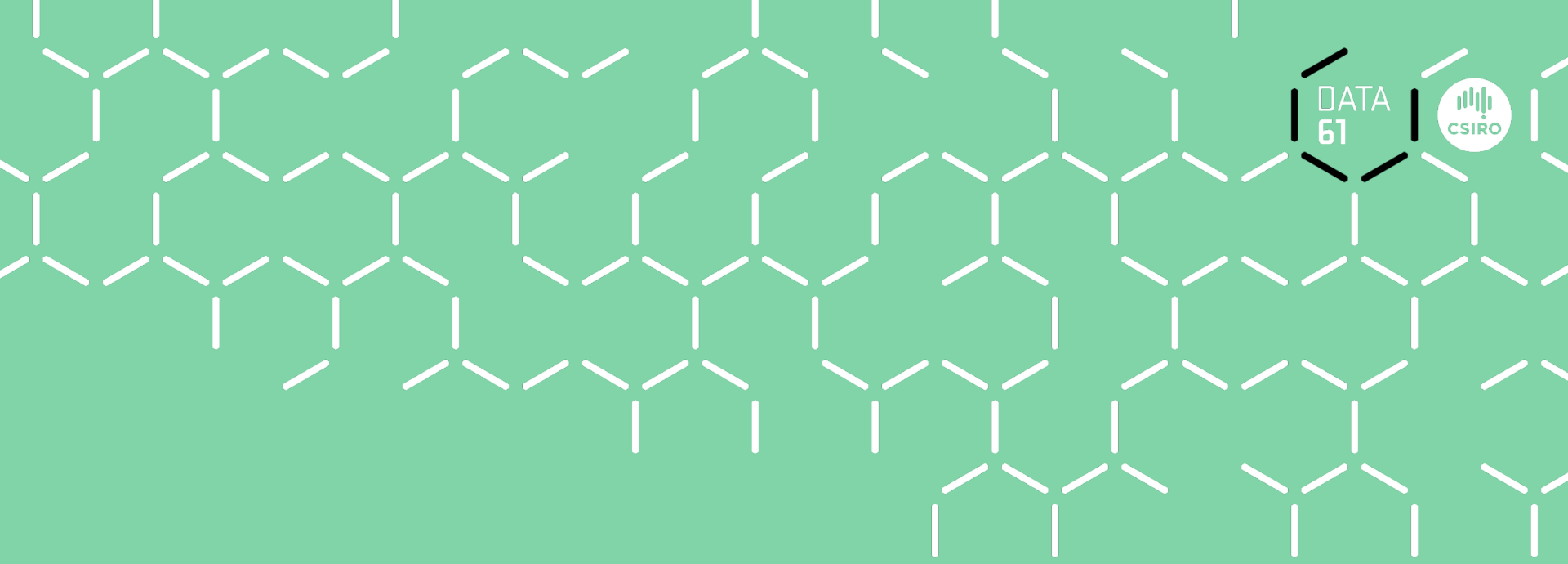
- TCB Object:
  - kernel's representation of a thread.
  - contains:
    - Caps: CSpace, VSpace, IPC Buffer Frame
    - Other: IP (instruction pointer), SP (stack pointer), IPC Buffer, Priority
- IPC Buffer
  - buffer used to pass data during IPC
    - 512 byte object, must be wholly in one frame
    - also used for all syscalls
  - passed to TCB as:
    - index to Frame cap in TCB's CSpace
    - address where it is mapped in TCB's VSpace

TCB: API

- Configure
  - `seL4_TCB_Configure`: set CSpace, VSpace, IPC buffer, priority
- Write Registers
  - `seL4_TCB_ReadRegisters`: retrieve current registers (ip, sp, etc.)
  - `seL4_TCB_WriteRegisters`: set the current registers (ip, sp, etc.)
  - Set of registers is arch-specific
  - Defined in arch-specific struct
- Resume
  - `seL4_TCB_Resume`: start thread running at it's current instruction pointer
- Suspend
  - `seL4_TCB_Suspend`: stop thread running

Hands-on: hello-2 (part 3)

- Goal: Create a thread
- TODO 6: get CSpace root Cnode
- TODO 7: get our VSpace root page directory
- TODO 8: create a new TCB
- TODO 9: initialise the new TCB
- TODO 10: give the new thread a name
- TODO 11: set instruction pointer
- TODO 12: set stack pointer
- TODO 13: actually write the TCB registers
- TODO 14: start the new thread running
- TODO 15: print something in new thread
- Wow – that's a lot of effort for a thread!



seL4 API and libraries
- IPC

IPC in seL4 – Endpoints ®

- Endpoint Object
  - Formerly: synchronous endpoint object
  - enables synchronous (blocking) communication
  - communicating threads must hold caps to same endpoint
- Endpoint Caps
  - ***master cap*** (typically receiver): received when creating the endpoint object
  - ***derived caps*** (senders): minted from master (or other) caps
  - ***badge***: identifies specific sender cap
  - ***reply cap***: temporary cap allows receiver to reply to sender for two-way communication
- Message Registers
  - Data to be sent is stored in message registers (MR)
  - Stored in machine registers or in IPC buffer

IPC – Endpoints: API ®

- Sending and Receiving
  - `seL4_Send`: send message registers. Blocks if receiver not Recving.
  - `seL4_Recv`: wait for a send on endpoint. Blocks if no send pending.
  - `seL4_Call`: send and recv in one syscall. Also sends a reply cap.
  - `seL4_Reply`: send a message using reply cap.
  - `seL4_ReplyRecv`: send and recv in one syscall. Using reply cap to send.
- Message Registers
  - `seL4_GetMR`: retrieve a given message register from IPC buffer.
  - `seL4_SetMR`: set a given message register in IPC buffer.
  - `seL4_GetCap`: retrieve a cap sent in an IPC.
  - `seL4_SetCap`: prepare a cap to send in an IPC.
- Tag: `seL4_MessageInfo_t`
  - Label, message length, number of caps, caps unwrapped

seL4 IPC – Notification ®

- Notification Object
  - Formerly: asynchronous endpoint object
  - allows one thread to send a notification to another
  - notification: asynchronous (non-blocking) message
- Notification Object Caps
  - master cap (receiver), derived caps (senders)
- API
  - `seL4_Signal`:
    - AORs sender badge
  - `seL4_Wait`:
    - Blocks if no new notification since last wait

seL4 API: CSpace

- CNode Object
  - consists of slots in which capabilities are stored
  - can also store CNode caps in slots, creates hierarchical CSpace structure
- API
  - insert cap:
    - indirectly: through `seL4_Untyped_Retype`
    - `seL4_CNode_Copy`: copy from one slot into another
    - `seL4_CNode_Mint`: copy but change some cap attributes
    - `seL4_CNode_Move`: remove from one slot and put in another
  - remove cap
    - `seL4_CNode_Delete`:remove cap from cslot, destroy object if last cap
    - `seL4_CNode_Revoke`: delete all child caps (e.g. through copy or retype)
    - `seL4_CNode_Recycle`: revoke and then reset object attributes

Cspace slots and addresses ®

- CSpace structure
  - Hierarchy of CNodes: 1-level, 2-level, 3-level, …
- CNode size (radix)
  - 2^{radix} slots
- CSpace Slot (CSlot) address
  - CPtr: 32-bit Word
  - CPtr is resolved based on CSpace structure (see examples)
  - CPtr resolution relative to a **root cnode**
  - **Depth:** how many bits of CPtr to resolve
- Use of CSlot addresses in syscalls:
  - Just CPtr (implicit root = TCB's CNode cap & implicit 32-bit depth)
  - Explicit root (is a 32-bit depth CPtr with implicit root) & CPtr (called index) & explicit depth

CSpace example and addressing

CNode 0: 2^{16}	
0	
1	A
2	U: cnode 1
3	V: cnode 0
4	
…	…

Cnode 1: 2^{16}	
0	
1	W: cnode 2
2	B
3	X: cnode 0
4	
…	…

Cnode 2: 2^{16}	
0	
1	
2	C
3	
4	Y: cnode 3
…	…

Cnode 3: 2^{16}	
0	
1	
2	D
3	Z: cnode 4
4	
…	…

Cnode 4	
0	
1	E
2	
3	
4	
…	…

- B: CPtr:
  - 0x 0002 0002
- W: CPtr:
  - 0x 0002 0001
- X: CPtr:
  - 0x 0002 0003
- A: CPtr:
  - 0x 0003 0001
- U: CPtr:
  - 0x 0003 0002

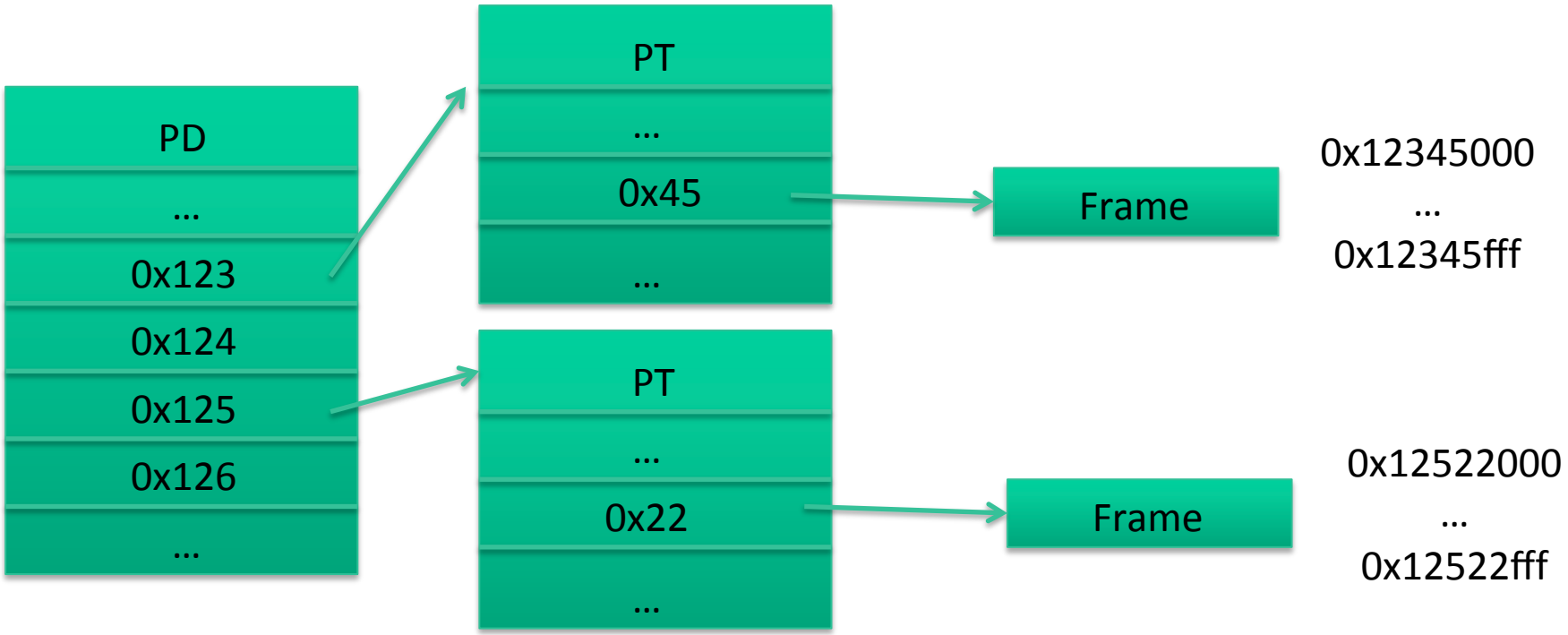
- C: Root index
  - Root: W index: 0x 0002 depth: 16
- Y: Root index
  - Root: W index: 0x 0004 depth 16
- D: Root index
  - root: W index: 0x 0004 0002 depth: 32
- Z: Root index
  - root: W index: 0x 0004 0003 depth: 32
- E: Root index
  - Impossible

CSpace and guards

- CNode caps can also have guards
  - Inspired by *guarded page tables*
  - Enables sparseness
- Guard
  - Fixed prefix of a CNode's address fragment
  - Example:
    - CNode: radix = 8, guard size = 8 bits value = 0x44
      - CNode's address fragment must be 0x44XX
    - CNode: radix = 12, guard size = 4 bits value = 0x8
      - CNode's address fragment must be 0x4XXX
    - 2-level CSpace: CNode 0: r: 8 g: 8,0x44 CNode 1: r: 12 g: 4,0x4
      - CPtrs must be 0x 44XX 4XXX

seL4 API: Vspace ®

- VSpace:
  - Objects: PageDir (PD), PageTable (PT), Frame
  - represents mapping: virtual address → physical address
    - i.e. abstraction of CPU page table
- Example (on ARM):



seL4 API: Vspace (contd.) ®

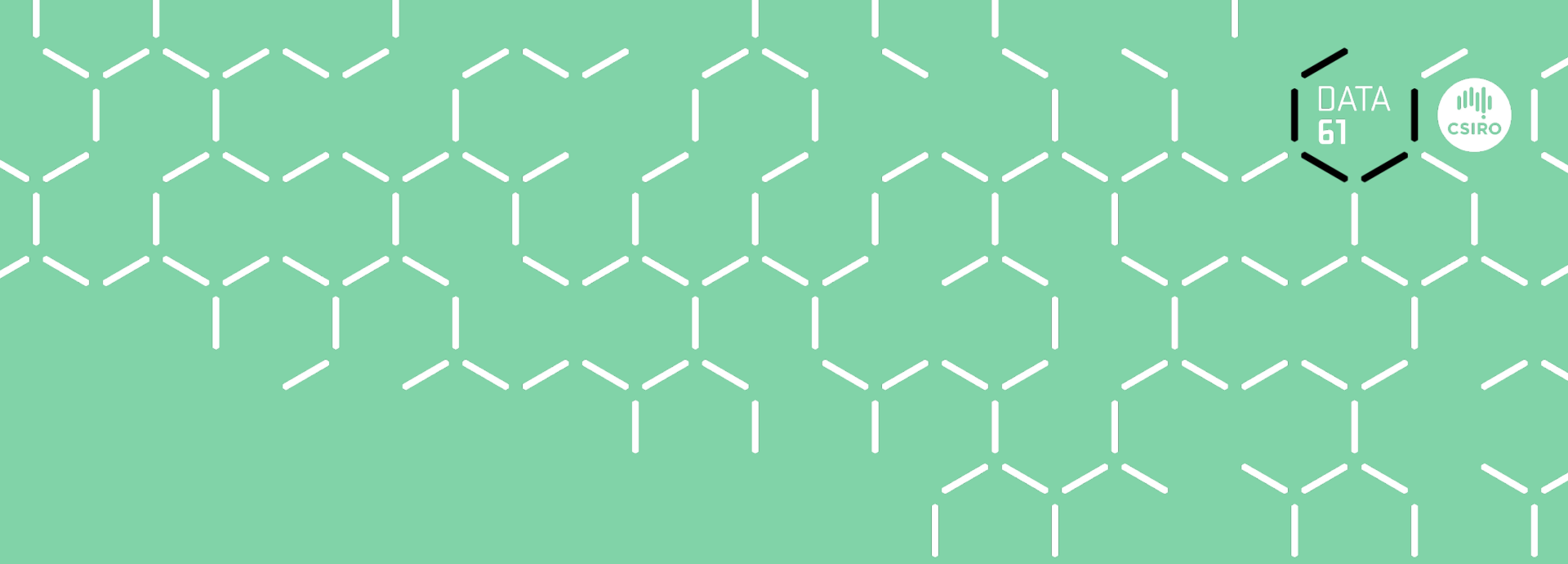
- VSpace-related objects are platform-specific
- ARM
  - PD: 16KiB, 4 byte slots (1MiB of address space/slot)
  - PT: 1KiB, 4 byte slots
  - Frame: 4KiB, 64KiB, 1MiB, 16MiB
- ARM HYP (uses LPAE long descriptors)
  - PD: 16KiB, 8 byte slots (2MiB of address space/slot)
  - PT: 4KiB, 8 byte slots
- x86
  - PD: 4KiB, 4 byte slots
  - PT: 4KiB, 4 byte slots
  - Frame: 4KiB, 4MiB

seL4 VSpace: API

- PD
  - `seL4_<ARCH>_PageTable_Map`
  - `seL4_<ARCH>_PageTable_Unmap`
  - Can map in large frames directly:
    - `seL4_<ARCH>_Page_Map`
    - `seL4_<ARCH>_Page_Unmap`
- PT
  - `seL4_<ARCH>_Page_Map`
  - `seL4_<ARCH>_Page_Unmap`
- Note:
  - <ARCH> is either ARM or IA32
  - Libsel4utils provides architecture independent versions of calls
    - E.g.: `seL4_ARCH_Page_Map`

Hands-on: hello-3

- Extends code-base from hello-2 to do IPC between threads
- Prepare
  - `make ia32_hello-3_defconfig`
  - `make`
  - Edit `apps/hello-3/src/main.c`
- Root task main thread:
  - TODO 1: get a frame cap for the ipc buffer
  - TODO 2: try to map the frame the first time
  - TODO 3: create a page table
  - TODO 4: map the page table
  - TODO 5: then map the frame in
  - TODO 6: create an endpoint
  - TODO 7: make a badged copy of it
  - TODO 8: set the data to send
  - TODO 9: send & wait for reply
  - TODO 10: get reply message
- Second thread:
  - TODO 11: wait for a message
  - TODO 12-13: get & check message
  - TODO 14-15: send message back



seL4 API and libraries
- Starting a new process

What's a seL4 process? ®

- seL4 doesn't have a concept of "process"
- Traditional process:
  - Each process has its own:
    - CSpace & VSpace & Threads
- Non-traditional process:
  - Different combinations of VSpace and CSpace sharing:
    - Shared CSpace, separate VSpaces
    - Separate CSpace, shared Vspace
    - Partially shared CSpace
    - Partially shared VSpace

Build system: non-root task apps

- Adding a non-root-task app, Steps:
  - Example: root task: hello-4, non root-task app: hello-4-app
  - Make a new app for it (e.g. hello-app-4), with config and src files
  - Add dependency information about the new app in root task app's Kbuild
    - `hello-4-components-y += hello-4-app`
    - `hello-4-components = $(addprefix $(STAGE_BASE)/bin/, $(hello-4-components-y))`
    - `hello-4: export COMPONENTS=${hello-4-components}`
    - `hello-4: ${hello-4-components-y} kernel_elf $(hello-4-y)`
  - And in the root task app's Makefile to add it to the cpio archive
    - {COMPONENTS}: false
    - archive.o: ${COMPONENTS} $(Q)mkdir -p $(dir $@) ${COMMON_PATH}/files_to_obj.sh $@ _cpio_archive $^

Library: vspace ®

- Interface for managing VSpaces
  - manage current VSpace
  - manage other VSpaces
    - note: create is not part of vspace API !
  - allocate frames and map them into a VSpace
- Key concepts
  - ***reservation***: portion of VSpace, that will not be given to others
  - ***mapping***: frame mapped into a VSpace at a virtual address
- Interfaces defined
  - `vspace`
- Implemented by
  - sel4utils

Vspace: API

- Files
  - libsel4vspace, #include <vspace/vspace.h>
- Datastructs
  - `vspace_t`
  - `reservation_t`: a reserved range of vspace addresses
- Functions
  - `vspace_reserve_range`, `vspace_free_reservation`
  - `vspace_new_pages`: create frames and map into VSpace
  - `vspace_map_pages`: map given frames into VSpace
  - `vspace_unmap_pages`: provides different ways to free frame object
  - `vspace_get_cap`: get frame cap for specific virtual address
  - `vspace_get_root`: get cap to PD of VSpace

Library: sel4utils ®

- Utility code to make life easier
  - create and manage threads and processes
  - create vspaces, implement vspace interface
  - load ELF code
- Key concepts
  - process: CSpace + VSpace + TCB
- Interfaces implemented
  - `vspace`
  - `sel4utils`: util functions provided by the library
    - Thread and process management
    - Logging and profiling
    - Architecture agnostic functions

seL4utils: API

- Files:
  - libseL4utils, #include <sel4utils/…>: vspace.h, process.h, mapping.h
  - `libs/libsel4utils/include/sel4utils`
- Datastructs
  - `process_t`
  - `thread_t`
- Functions
  - `sel4utils_bootstrap_vspace_with_bootinfo`
  - `sel4utils_get_vspace`: create new vspace
  - `sel4utils_configure_process`: create process
  - `sel4utils_spawn_process_v`: start process
  - `sel4utils_*_cap_to_process`: mint, copy
  - `seL4_ARCH_*`: architecture independent wrapper for seL4 syscalls

Hands-on: hello-4 (part 1)

- Create a new process
- Prepare
  - `make ia32_hello-4_defconfig`
  - `make`
- Edit `apps/hello-4/src/main.c`
  - TODO 1: create a vspace object
  - TODO 2-3: use sel4utils to make a new process
  - TODO 6: spawn the process
- Edit `apps/hello-4-app/src/main.c`
  - Fix it so that it doesn't fail
- Build and Run
  - New process should print something

Hands-on: hello-4 (part 2)

- Add IPC between the root task and the new process
- Edit `apps/hello-4/src/main.c`
  - TODO 4: make a cspacepath for the new endpoint cap
  - TODO 5: copy the endpoint cap and add a badge to the new cap
  - TODO 7: wait for a message
  - TODO 8: send the modified message back
- Edit `apps/hello-4-app/src/main.c`
  - TODO 9: send and wait for a reply
- Build and run
  - See message sent back and forth!

What’s Next?

Advanced seL4

- In-depth VSpace and CSpace
  - See seL4 manual: http://sel4.systems/Info/Docs/seL4-manual-latest.pdf
- Reimplement hello-2 (and 3 and 4) using only seL4 API
- Device drivers (in seL4; or with CAmkES)
  - Libplatsupport: https://github.com/seL4/util_libs/libplatsupport
  - Libsel4platsupport: https://github.com/seL4/sel4_libs/libsel4platsupport
- CapDL: capability distribution language – static system setup
  - https://github.com/seL4/capdl/capDL-tool
  - https://github.com/seL4/capdl/capdl-loader-app
- A whole operating system (e.g. UNSW Advanced Operating Systems Assignments)
  - http://www.cse.unsw.edu.au/~cs9242
  - https://bitbucket.org/kevinelp/unsw-advanced-operating-systems