\Orchestrating a brighter world NEC

A Journey through Unikraft's Build System

Simon Kuenzer <simon.kuenzer@neclab.eu>
Senior Researcher, NEC Laboratories Europe GmbH

Xen Summit 2019, Chicago

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreements no. 675806 ("5G CITY"). This work reflects only the author's views and the European Commission is not responsible for any use that may be made of the information it contains.

5GCity

Orchestrating a brighter world

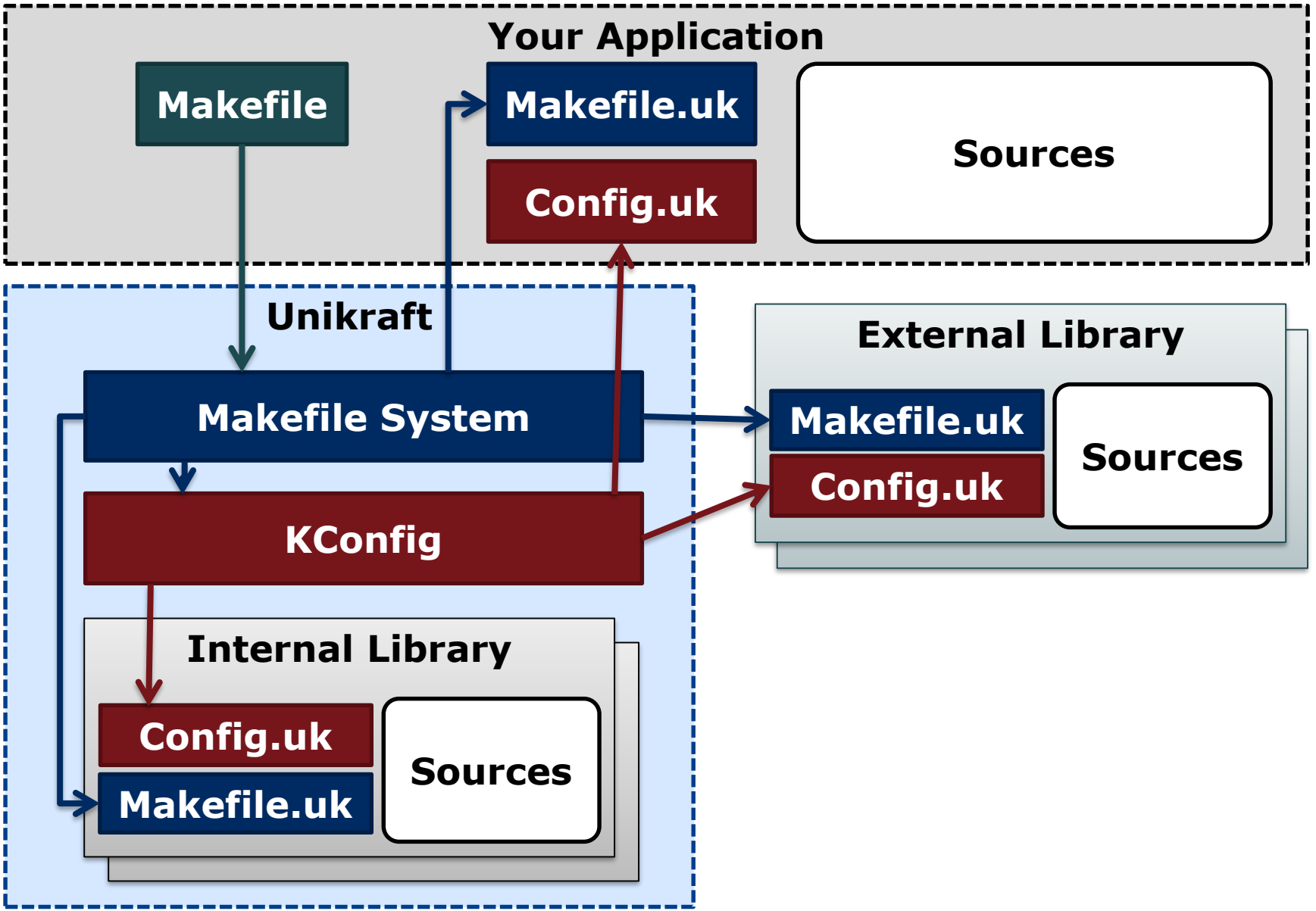
NEC brings together and integrates technology and expertise to create the ICT-enabled society of tomorrow.
We collaborate closely with partners and customers around the world, orchestrating each project to ensure all its parts are fine-tuned to local needs.

Every day, our innovative solutions for society contribute to greater safety, security, efficiency and equality, and enable people to live brighter lives.

Unikraft's Build System

Overview

Build System: Loading & Parsing



Unikraft's 3 Build Stages

(1) Fetch

- Download and decompress external sources
  e.g., a library hosted on GitHub, Sourceforge
- Patch downloaded files

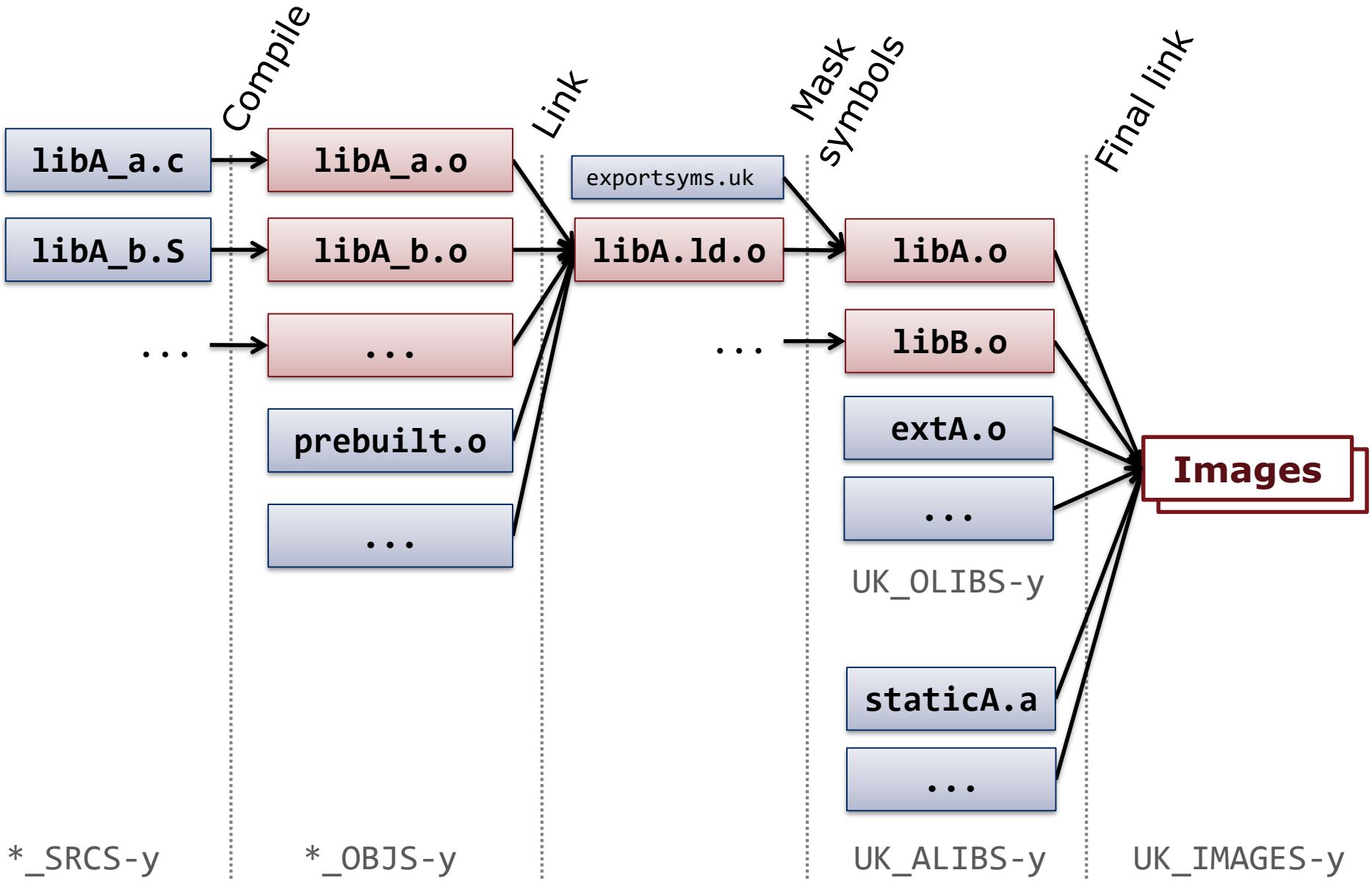
(2) Prepare

- Further preparation steps to the sources, for instance:
  - Call `./configure` of downloaded library sources
  - Generate further sources or headers required for building

(3) Compile & Link

- Compile sources
- Link libraries
- Link final images

Build Stage 3: Compiling and Linking



Make parameters

Unikraft's base Makefile

```
$ make A=[APP] L=[LIBRARIES] P=[PLATFORMS] V=[1/0] [target]
```

Parameter	Description
`A=[APP]`	Path to application directory
`L=[LIBRARIES]`	Colon-separated list of paths to external libraries
`P=[PLATFORMS]`	Colon-separated list of paths to external platform libs
`V=[1/0]`	Verbose mode (on/off)
`[target]`	**Build target**
`help`	Show overview of targets
`menuconfig`	Configure and select target images (default when there is no `.config`)
`all/images`	Build everything (default) *`+libs`*
`libs`	Build libraries *`+prepare`*
`prepare`	Run preparations steps *`+fetch`*
`fetch`	Download, extract, and patch external code

Unikraft Libraries

Integrate *own* libraries and applications

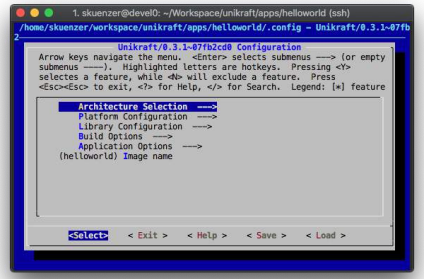
Libraries/Applications/Platforms: Necessary files

`Makefile` **applications only*

- Invoke Unikraft build for simplification

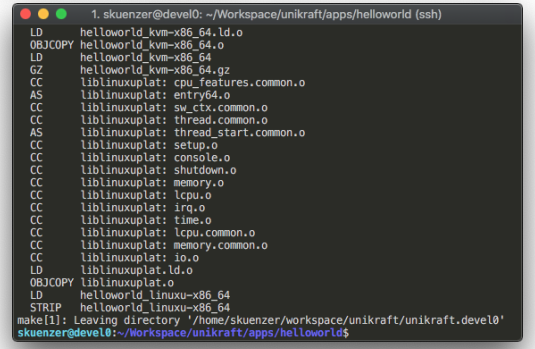
`Config.uk`

- Configuration options
  - Settings saved as part of `.config`
- Specifying library dependencies and depending options



`Makefile.uk`

- Registration to the build system
- Specification of source files
- Extra custom Make rules
  - For instance for preparing the sources



`exportsyms.uk`

- Masking of symbols

`Linker.uk` **platform libraries only*

- Platform-dependent rules for linking final image

Makefile **applications only*

- Simplify application building
- Changes to Unikraft base directory and invokes make

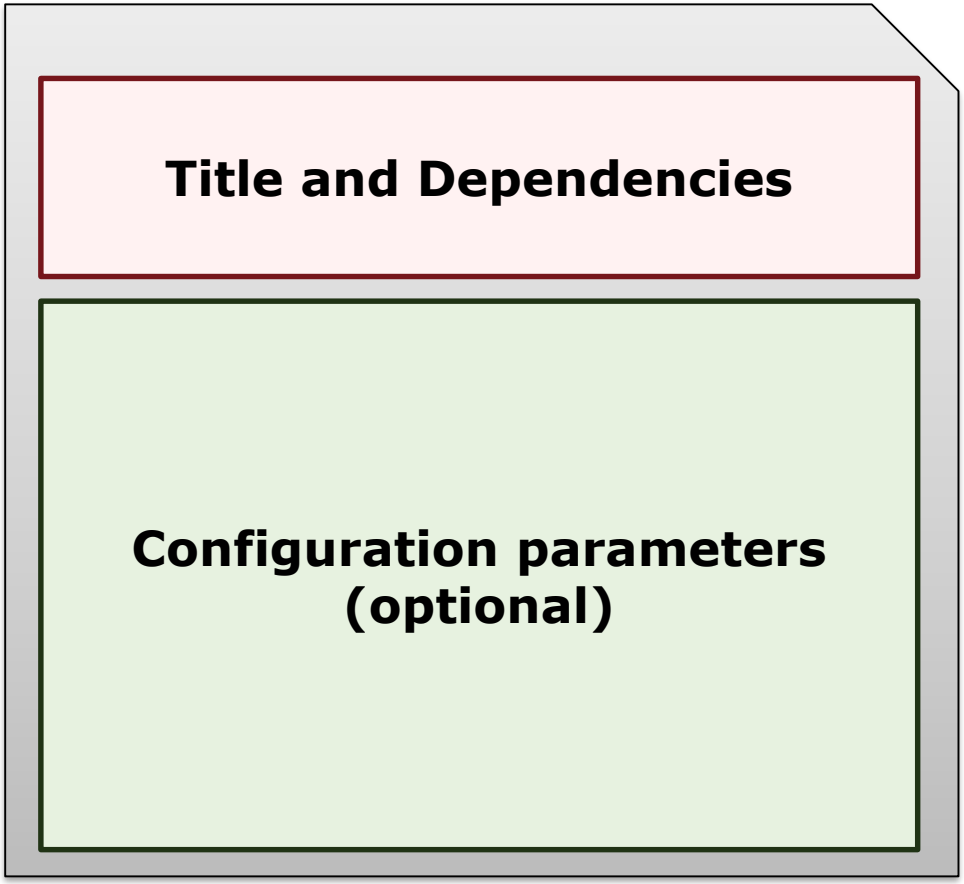
```
UK_ROOT   ?= $(PWD)/../../unikraft
UK_LIBS   ?= $(PWD)/../../libs
LIBS      := $(UK_LIBS)/libA:$(UK_LIBS)/libB

all:
	@$(MAKE) -C $(UK_ROOT) A=$(PWD) L=$(LIBS)

$(MAKECMDGOALS):
	@$(MAKE) -C $(UK_ROOT) A=$(PWD) L=$(LIBS) (MAKECMDGOALS)
```

Config.uk

- KConfig syntax[1]
- Structure:



[1] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

Config.uk: Title and Dependencies for Libraries

```
menuconfig LIBMYLIB
        bool "ukmylib: my scheduler“
        ### libraries are off as default
        default n
        ### dependencies
        select LIBNOLIBC if !HAVE_LIBC

if LIBMYLIB
        ### list of configuration parameters goes here
endif
```

- `menuconfig` defines a submenu for the following configuration options
  - *Note: Use just* `config` *without following* `if`*-block when the library does not have any configuration parameters*
- `select` keyword is used to describe dependencies to other libraries
  - Supports conditional expressions[1]
  - Multiple `select` lines are possible for a single configuration

Config.uk: Dependencies for Applications

```
### Invisible option (no description) that is set as
### default to ‚y‘ in order to select dependencies
config LIBMYAPP
        bool
        default y
        select LIBNOLIBC if !HAVE_LIBC

### list of configuration parameters goes here
```

- Invisible `bool config` to select dependencies
  - Applications are not defining an own submenu. Because just a single application can currently be selected for one Unikernel build, applications are enabled as default
- Like for libraries, `select` keyword is used to describe dependencies to libraries

Config.uk: Configuration Parameter

```
config LIBMYLIB_SETTING
        [type] "[description]"
        default [value]
        select LIBOTHER
```

- Name-space configuration options!
  - Prepend library name in front of parameters: *here:* `LIBMYLIB_`
- Configurations will appear as `CONFIG_[CONFIGNAME]` in the build system and in the sources (include `uk/config.h`) (*here:* `CONFIG_LIBMYLIB_SETTING`)
- `[type]` can be one of bool, int (unsigned), hex, string:

	Makefile.uk	#include <uk/config.h>
bool	y/n	„y" is defined as 1 „n" is not defined
int	Hexadecimal	Defined as hexadecimal
hex	Hexadecimal	Defined as hexadecimal
string	String	Defined as String

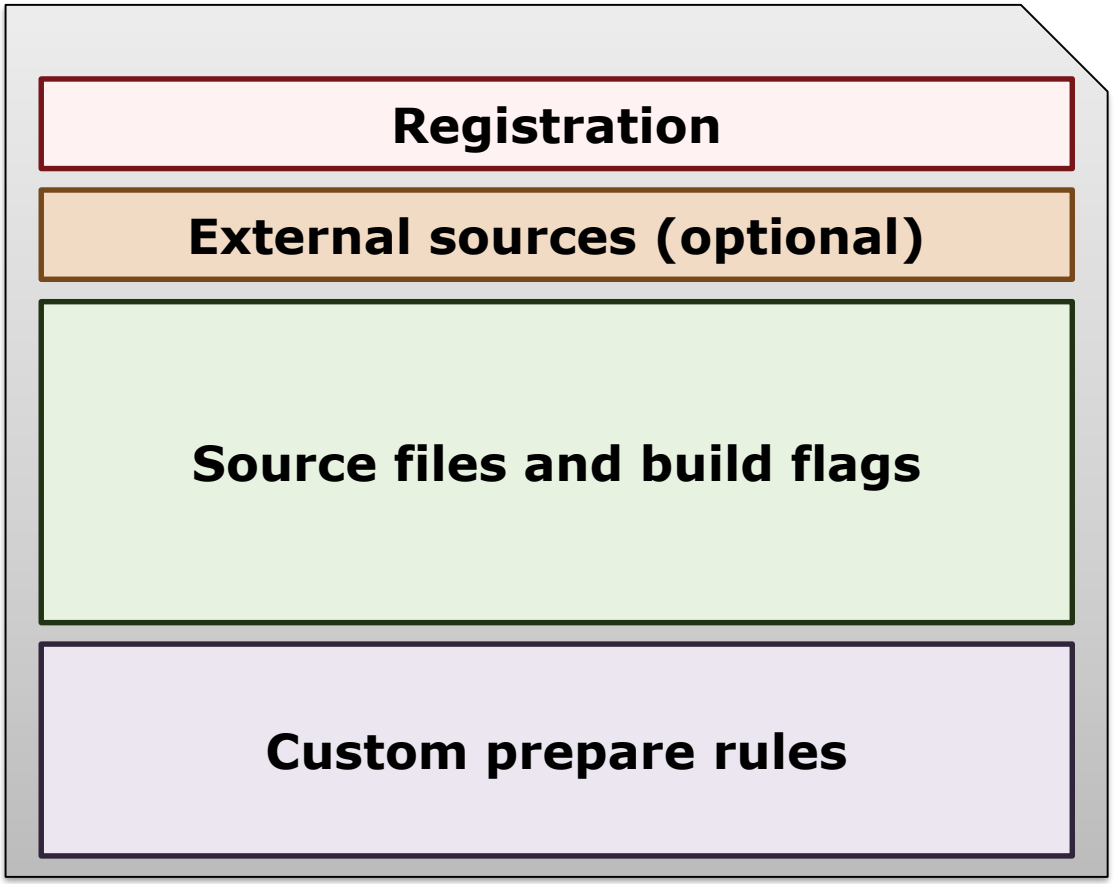
- Advanced options, like choice lists, are documented at: https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

Makefile.uk

Makefile syntax

- Unikraft provides helper functions and variables
- Unikraft expects specific variables to be filled

Structure



Makefile.uk: Registration

Registration with `addlib` / `addlib_s` helpers[1]

- The first thing that has to be done in a `Makefile.uk`
- Libraries (depending on being enabled)
  Replace `[libmylib]` and `CONFIG_LIBMYLIB` accordingly:

```
$(eval $(call addlib_s,[libmylib],$(CONFIG_LIBMYLIB)))
```

- Applications
  Replace `[libmyapp]` accordingly:

```
$(eval $(call addlib,[libmyapp]))
```

- The namespace for variables is defined by application/library name
  here: prefixed in uppercase: `LIBMYLIB_`, `LIBMYAPP_`
- The following variables are populated after the call

`*_BASE`	Path to library folder
`*_BUILD`	Path to library's output/build folder *Note: Place all generated files during building in here, never in the base*

- The following overrides are available after the call

`*_EXPORTS`	Path to an alternative `exportsyms.uk` (optional)

[1] `support/build/Makefile.rules`

Makefile.uk: External sources (optional)

- Download one archive with (additional) sources and extract them with `fetch` / `fetchas` helpers[1]
  - Example with lwIP:

```
LIBLWIP_ZIPNAME=lwip-2.1.2
LIBLWIP_URL=http://download.savannah.nongnu.org/releases/lwip/$(LIBLWIP_ZIPNAME).zip
$(eval $(call fetch,liblwip,$(LIBLWIP_URL)))
```

  - `.tar.gz`, `.tgz`, `.tar.xz`, `.txz`, and `.zip` are currently supported[1]
  - The following variables are populated after this call

`*_ORIGIN`	Path to folder containing extracted archive files

- If some downloaded source files need to be patched, use `patch` helper[1]:
  - Example with lwIP:

```
LIBLWIP_PATCHDIR=$(LIBLWIP_BASE)/patches
$(eval $(call patch,liblwip,$(LIBLWIP_PATCHDIR),$(LIBLWIP_ZIPNAME)))
```

  - This command applies all patches found in `$(LIBLWIP_PATCHDIR)` to the sub-directory `$(LIBLWIP_ZIPNAME)` of the previously extracted sources

[1] `support/build/Makefile.rules`

Makefile.uk: Source files

Full paths to source files are added to the *_SRCS-y list:

```
# Source file from library directory
LIBMYLIB_SRCS-y += $(LIBMYLIB_BASE)/source.c

# Source file from extracted archive
LIBMYLIB_SRCS-y += $(LIBMYLIB_ORIGIN)/another_src.c
```

- Compile rule and target is automatically generated by build system based on file extension:
  - Currently supported: `.S`, `.sx`, `.s`, `.c`, `.C`, `.cc`, `.cp`, `.cxx`, `.cpp`, `.CPP`, `.c++`, `.lds.S`
  - .o-binary is created within the library output directory based on the source filename:

```
$(LIBMYLIB_BASE)/source.c.       → $(LIBMYLIB_BUILD)/source.o
$(LIBMYLIB_ORIGIN)/another_src.c → $(LIBMYLIB_BUILD)/another_src.o
```

    - Note: Only the filename without extension matters for the target file name. The source file extension and path is irrelevant. In conflicting cases, use variants (see next slide).
- The following include and build flag lists apply for a source file:

`CFLAGS-y`	Global build flags (*here*: C sources, equivalents for other types exist)
`CINCLUDES-y`	Global includes (*here*: C sources, equivalents for other types exist)
`*_CFLAGS-y`	Library-internal build flags (*here*: C sources, equivalents exist)
`*_CINCLUDES-y`	Library-internal includes (*here*: C sources, equivalents exist)
`*_[FILENAME]_FLAGS`	Source file specific build flags
`*_[FILENAME]_INCLUDES`	Source file specific includes

Makefile.uk: Source File Variants

Variants exist because of two reasons:

- Conflicting output file names (previous slide)
- Necessity to compile a single source file multiple times with different flags (e.g., newlib *scanf() variants)

Variant names are added with a pipe symbol after source:

```
LIBMYLIB_SRCS-y += $(LIBMYLIB_BASE)/source.c
LIBMYLIB_SRCS-y += $(LIBMYLIB_BASE)/source.c|variant
LIBMYLIB_SRCS-y += $(LIBMYLIB_ORIGIN)/source.c|origin
```

- …produces:

```
$(LIBMYLIB_BASE)/source.c          → $(LIBMYLIB_BUILD)/source.o
$(LIBMYLIB_BASE)/source.c|variant  → $(LIBMYLIB_BUILD)/source.variant.o
$(LIBMYLIB_ORIGIN)/source.c|origin → $(LIBMYLIB_BUILD)/source.origin.o
```

- Variants have their own specific build flags and includes:

*_[FILENAME]_FLAGS	Build flags for source file without variant specification
*_[FILENAME]_INCLUDES	Includes for source file without variant specification
*_[FILENAME]_[VARIANT]_FLAGS	Build flags for source file for variant [VARIANT]
*_[FILENAME]_[VARIANT]_INCLUDES	Includes for source file for variant [VARIANT]

Makefile.uk: Externally Compiled Sources

- Unikraft supports including externally compiled sources
  Cases where it may happen:
  - Code only available as binary form
  - Compiling is done by different build system
    (e.g., invoked by custom prepare rules)

- Depending on the type, various places exist to add them:
  - `.o`-object files are added to the `*_OBJS-y` list:

```
LIBMYLIB_OBJS-y += $(LIBMYLIB_BASE)/prebuilt.o
```

  - `.o`-libraries are registered to the global list `UK_OLIBS`
    (remember to use the library switch `CONFIG_*`):

```
UK_OLIBS-$(CONFIG_LIBMYLIB) += $(LIBMYLIB_BASE)/prebuilt_lib.o
```

  - Static libraries are registered to the global list `UK_ALIBS`
    (remember to use the library switch `CONFIG_*`):

```
UK_ALIBS-$(CONIG_LIBMYLIB) += $(LIBMYLIB_BASE)/static_lib.a
```

  - *Note: Shared libraries (`.so`) are currently not supported*

Makefile.uk: Scope of Headers (Includes)

- Global headers (e.g., library API)
  (remember to use the library switch `CONFIG_*`):

```
CINCLUDES-$(CONFIG_LIBMYLIB) += -I$(LIBMYLIB_BASE)/include/api
```

- Library-Internal headers

```
LIBMYLIB_CINCLUDES-y += -I$(LIBMYLIB_BASE)/include/internal
```

- File-specific headers

```
# mysrc.c
LIBMYLIB_MYSRC_INCLUDES      += -I$(LIBMYLIB_BASE)/include/mysrc

# Variant var0 of mysrc.c: mysrc.c|var0
LIBMYLIB_MYSRC_VAR0_INCLUDES += -I$(LIBMYLIB_BASE)/include/mysrc
```

- Equivalent to this, you can set build flags within a specific scope
  - `CFLAGS-$(CONFIG_LIBMYLIB), LIBMYLIB_CFLAGS-y, LIBMYLIB_MYSRC_FLAGS, LIBMYLIB_MYSRC_VARIANT_FLAGS`

Makefile.uk: Custom Prepare Rules

Reason

- Generate files (headers, sources) needed for build
- Invoke parts of ported library build system, like `./configure`

Defined as custom Make rules

- Use `build_cmd` to prettify the output[1]
  (in cases where `build_cmd` is not applicable use `verbose_cmd`)

```
$(LIBMYLIB_BUILD)/generated.h: [dependencies]
   $(call build_cmd,NM,libmylib,$@,$(NM) -n $(LIBMYLIB_BASE)/symtab.in > $@)
```

If used, set marker of fetch stage as dependency

- Download marker: `$(LIBMYLIB_BUILD)/.origin`
- Patched marker: `$(LIBMYLIB_BUILD)/.patched`

Register generated files to prepare stage

```
UK_PREPARE-$(CONFIG_LIBMYLIB) += [generated file/phony]
```

[1] `support/build/Makefile.rules`

exportsyms.uk

- Re-masks the scope of each symbol of a library
  - Re-defines for each symbol if it is available for final linking
  - Intended to reduce potential clashing of symbols

- List of symbol names that should be available globally for final linking. Non-listed symbols become private to the library.
  - *Example (libnolibc):*

```
asprintf
vasprintf

# comments are ignored
opterr
optind
optopt
optreset
optarg
getopt
[...]
```

- Note: The build system will throw a warning when no `exportsyms.uk` file is provided
  - The scope of each library symbol stays unchanged in such a case

Best Practices

Best Practices

Porting existing libraries/applications is a challenging task

- Existing sources often only fit to their build and configuration system
- Often not intended to run on something else than Linux (assumptions to the OS)

If possible, compile all sources with Unikraft

- Including external build binaries is risky
  - Build flags may be incompatible (e.g., register usage/calling convention, LTO)
  - Mismatch of depending libraries (external vs. Unikraft's version)

Learn from existing build system

- Extract list of source files and build flags when compiling with original build system
- Study steps that generate files needed for the build
  - Try to run `./configure` with settings fitting to Unikraft environment
  - It is also possible to call `./configure` from Unikraft as prepare step

Provide initial stubs for missing symbols

- Completing compiling & linking (but not running) first,
  helps to get an better overview of missing functionality in Unikraft


Orchestrating a brighter world
NEC