<h1>MirageOS</h1>

Unikernels in OCaml

Me: Antonin Decimo, Tarides
@Rucikir or @MisterDA online

Slides from Romain Calascibetta
@dinosaure

OCaml

https://ocaml.org/

A functional, statically typed, general purpose, expressive, garbage-collected, programming language (also with objects)

```
# let square x = x * x
val square : int -> int = < fun >
# square 3
- : int = 9
# let rec fac x =
if x <= 1 then 1 else x * fac (x - 1)
val fac : int -> int = < fun >
# fac 5
- : int = 120
# square 120
- : int = 14400
```

<h3>The MirageOS project</h3>

The MirageOS project is mainly 3 things:

- An ecosystem
- A tool
- Multiple ABIs

Used to deploy Unikernels

- Specialized machine image
- Library operating system
- Deployed to cloud or ecosystem

<h3>Your MirageOS project</h3>

```
mirage/ mirage-tcpip  ocaml-git    mrmime       bechamel
      / colombe       duff         ocaml-base64 ocaml-x509
      / digestif      encore       ocaml-hex    happy-eyeballs
      / decompress    docteur      ke           ocaml-pgp
      / mirage-crypto paf-le-chien ocaml-rpc    prometheus
      / ocaml-cohttp  irmin        conan        ocaml-cstruct
      / ocaml-matrix  docteur      eqaf         bloomf
      / ocaml-tls     optint       ca-certs-nss ...
```

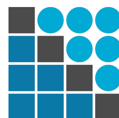
```ocaml
module Make (_ : _) ... = struct
  let start _ ... : unit Lwt.t =
    my_super_application ()
end
```

Kernel Virtualization Machine

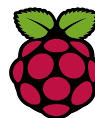
Xen

VIRTual Input Output

SeCure COMPuting mode



Raspberry PI 4



Muen

UNIX

<h3>3 degrees of freedom</h3>

- Let the user build their own service
- Choose which **implementation** will be used to concretize required *devices* by your application
- Choose which *target* you want to **deploy** you application to

<h3>A *device*</h3>

A *device* is a **specification** (an OCaml module signature) which can **interact** with a physical components of a computer. It can be:

```
module type KV = sig
  type t
  type key and value

  val read : t -> key -> value Lwt.t
  val write : t -> key -> value -> unit Lwt.t
end
```

A *KV*-store like a file-system

```
module type FLOW = sig
  type t

  val read : t -> bytes -> [ `Eof | `Data of int ] Lwt.t
  val write : t -> string -> unit Lwt.t
  val close : t -> unit Lwt.t
end
```

An *ongoing* connection with a peer

```
module type CLOCK = sig
  type t

  val now : t -> int64 Lwt.t
end
```

A clock

```
module type CONSOLE = sig
  type t

  val log : ('a, Format.formatter, unit, unit Lwt.t) format4 -> 'a
end
```

A console to *print* messages

<h3>A higher *implementation* from lower *devices*</h3>

An implementation which **provides** a *device* (a **specification**) can require multiple *devices*:

```
module Make_LITTLEFS : KV =
  functor (Block : BLOCK) ->
  functor (Posix_clock : PCLOCK) ->
  struct ... end
```

```
module Make_TCPIP : FLOW =
  functor (Time : TIME) ->
  functor (Random : RANDOM) ->
  functor (Netif : NETIF) ->
  functor (Ethernet : ETHERNET) ->
  functor (Arp : ARP) ->
  functor (Ip : IP) ->
  functor (Monotonic_clock : MCLOCK) ->
  struct ... end
```

```
module Make_TLS : FLOW =
  functor (Flow : FLOW) ->
  struct ... end
```

<h3>A higher *implementation from* lower *devices*</h3>

An implementation which **provides** a *device* (a **specification**) can require multiple *devices*:

```
module Make_LITTLEFS : KV =
  functor (Block : BLOCK) ->
  functor (Posix_clock : PCLOCK) ->
  struct ... end
```

```
module Make_TCPIP : FLOW =
  functor (Time : TIME) ->
  functor (Random : RANDOM) ->
  functor (Netif : NETIF) ->
  functor (Ethernet : ETHERNET) ->
  functor (Arp : ARP) ->
  functor (Ip : IP) ->
  functor (Monotonic_clock : MCLOCK) ->
  struct ... end
```

```
module Make_TLS : FLOW =
  functor (Flow : FLOW) ->
  struct ... end
```

unikernel.ml

```
module Make
  (Console : CONSOLE)
  (Flow : FLOW)
  (Store : KV) = struct
  let start console flow store =
    my_super_application console flow store
end
```

<h3>Resolve & Compose everything</h3>

opam tries to solve all requirements of devices.
Functoria composes implementations/devices from the given solution.

```
module Make
  (Console : CONSOLE)
  (Flow : FLOW)
  (Store : KV) = struct
  let start console flow store =
    my_super_application console flow store
end
```

unikernel.ml

```
let unikernel =
  foreign "Unikernel.Make"
    (console @-> flow @-> kv @-> job)

let my_flow = with_tls tcpip_flow

let () =
  register "my_super_application"
    [ unikernel $ default_console $ my_flow $
littlefs ]
```

config.ml

As a resolver

```
name: "my_super_application"
depends: [
  "mirage-console-target"
  "mirage-crypto-rng-mirage"
  "mirage-mclock-target"
  "mirage-pclock-target"
  "mirage-block-target"
  "mirage-time-lwt"
  "mirage-netif-target"
  "mirage-tcpip"
  "ocaml-tls"
  "littlefs"
]
```

my_super_application.opam

```
module Console = Mirage_console_target
module Random = Mirage_crypto_rng
module Monotonic_clock = Mirage_monotonic_clock_target
module Posix_clock = Mirage_posix_clock_target
module Block = Mirage_block_target
module Time = Mirage_time_lwt
module Netif = Mirage_netif

module My_ethernet = Make_ETHERNET(...)
module My_ARP = Make_APR(...)
module My_IP = Make_IP(...)
module My_TCPIP = Make_TCPIP
  (Time)
  (Random)
  (Netif)
  (My_ethernet)
  (My_ARP)(My_IP)
  (Monotonic_clock)
module My_flow = Make_TLS(TCPIP_Flow)
module My_KV = Make_LITTLEFS(Block)(Posix_clock)

include Make (Console) (My_flow) (My_KV)

let () =
  let console = Console.connect () in
  let random = Random.connect () in
  let mclock = Monotonic_clock.connect () in
  let pclock = Posix_clock.connect () in
  let block = Block.connect () in
  ...
  let littlefs = My_KV.connect block pclock in
  ... start console my_flow littlefs
```

main.ml

<h3>Link everything</h3>

Fetch and **compile** dependencies.
Compile and **link** application and dependencies to a final **object** file.

```
mirage_console_target.cmxa
mirage_crypto_rng_mirage.cmxa
mirage_mclock_target.cmxa
mirage_pclock_tartget.cmxa
mirage_block_target.cmxa
mirage_time_lwt.cmxa
mirage_netif_target.cmxa
mirage_tcpip.cmxa
ocaml_tls.cmxa
littlefs.cmxa
   +    transitive dependencies
```

Compiled libraries

```
unikernel.cmx
main.cmx
```

Compiled application

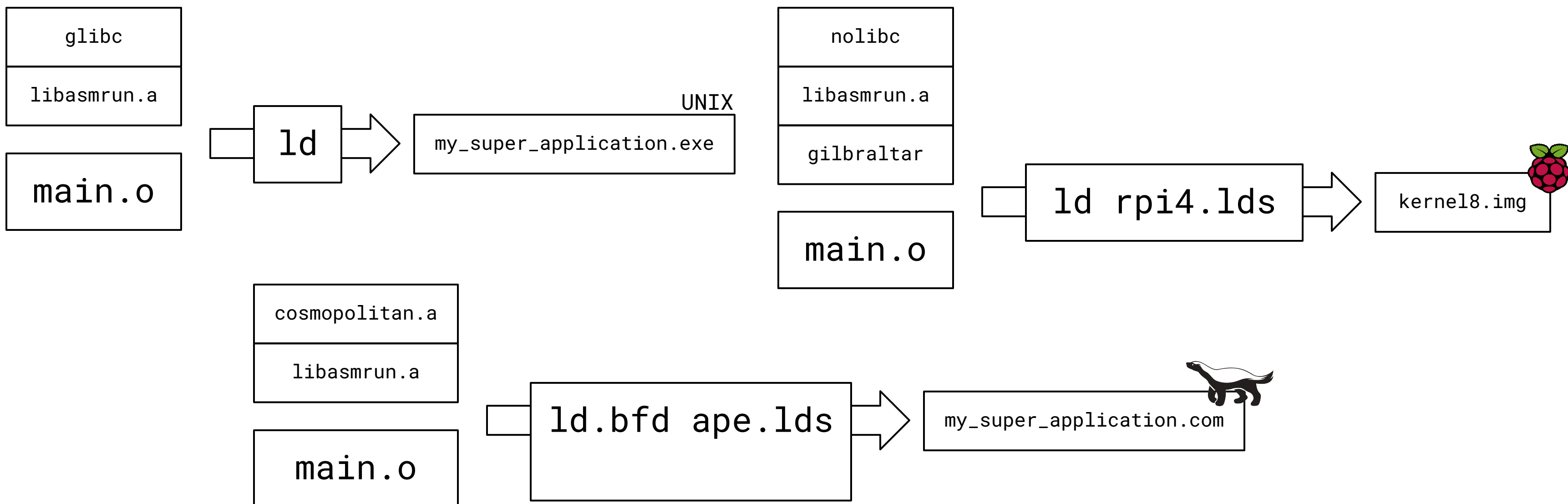
As a compiler

One and unique statically linked **object** file **main.o**

<h3>A *target*</h3>

A *target* is an ABI defined by:

- An **object** which provides few *functions*/***syscalls*** to interact with the system and the **caml runtime** (a *micro-kernel* or a *startup* **object** file)
- A **link script** to well-craft the final artifact (to an OS, an executable, etc.)
- A *toolchain* as a **coherent** context to build the final artifact

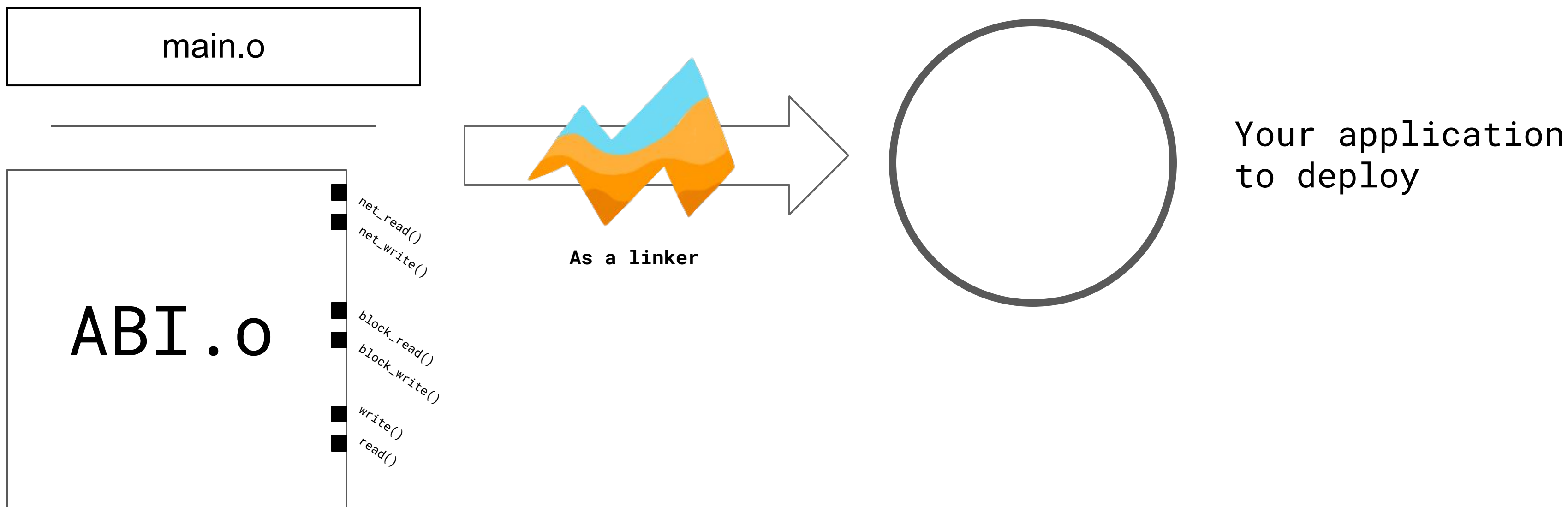


<h3>Craft everything(?)</h3>

Follow instructions to craft your application with the chosen ABI.

(the chosen ABI **implements** final and concrete functions to interact with your computer)

Link everything into your final artifact!



<h3>The mirage tool</h3>

```
mirage/ mirage-tcpip  ocaml-git    mrmime       bechamel
      / colombe       duff         ocaml-base64 ocaml-x509
      / digestif      encore       ocaml-hex    happy-eyeballs
      / decompress    docteur      ke           ocaml-pgp
      / mirage-crypto paf-le-chien ocaml-rpc    prometheus
      / ocaml-cohttp  irmin        conan        ocaml-cstruct
      / ocaml-matrix  docteur      eqaf         bloomf
      / ocaml-tls     optint       ca-certs-nss ...
```

```
module Make (_ : _) ... = struct
  let start _ ... : unit Lwt.t =
    my_super_application ()
end
```

As a resolver

As a compiler

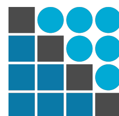
As a linker

Kernel Virtualization Machine

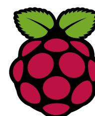
Xen

VIRTual Input Output

SeCure COMPuting mode



Raspberry PI 4

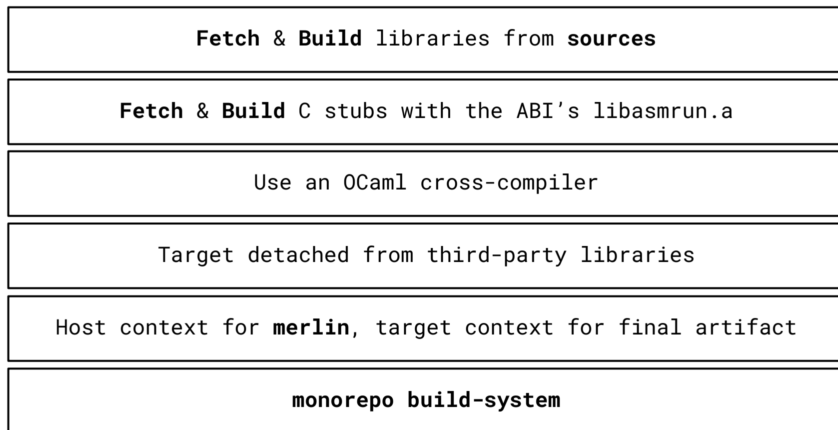


Muen

UNIX

<h3>A *coherent* tool / A *coherent* context</h3>

Coherent context for the **user**, the **libraries** and the **ABIs**.



<h3>Applications</h3>

https://github.com/roburio/dns-primary-git
https://github.com/roburio/dns-letsencrypt-secondary
https://github.com/yomimono/url-shortener
https://github.com/renatoalencar/ocaml-socks-client
https://github.com/roburio/tlstunnel
https://github.com/dinosaure/cri
https://github.com/palainp/mirage-sshfs
https://github.com/mirage/qubes-mirage-firewall
https://github.com/roburio/unipi : static website from a Git repository
https://github.com/dinosaure/contruno : a TLS termination proxy
https://github.com/mirage/dns-resolver : a DNS resolver that only trusts root servers
https://github.com/dinosaure/ptt : a full SMTP stack

A case for MirageOS and community networks?

Self-hosted, secure and simple services

PHP, Java, Bash no more!

Easy to deploy, restart