MirageOS

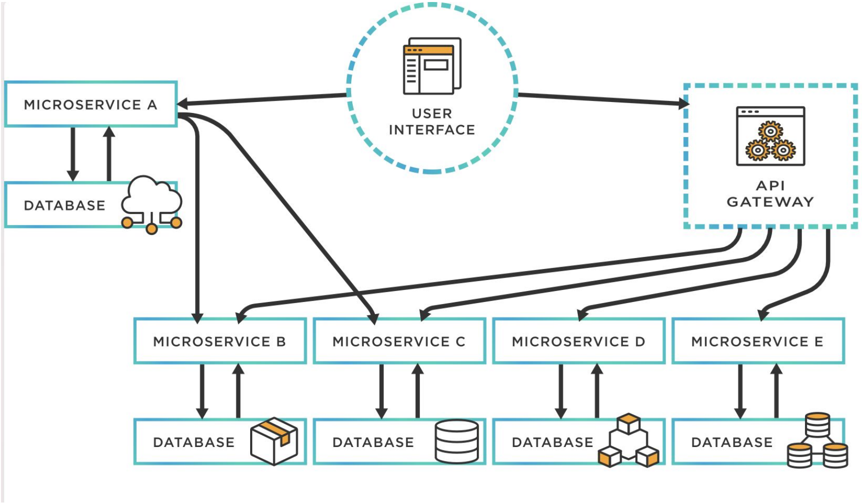
Building Custom Kernels with Library Operating Systems

Why Operating Systems: The Traditional Story

- Computers = **super** expensive
- We need to manage multi-user access to shared hardware with many processes
- Stable interfaces (networking, graphics, etc) to develop software
- Kernel/Userland Model
    - If something goes wrong in userland, we can have kernel sort things out
    - Ensures users and processes can't monopolize in-demand resources

Why Operating Systems: Today

- Is this still applicable?
    - Yes!
    - But not always
- Modern Web
    - Monoliths -> “Microservices”
    - Containers and virtualization galore!
    - Emphasis on flexibility and scalable systems



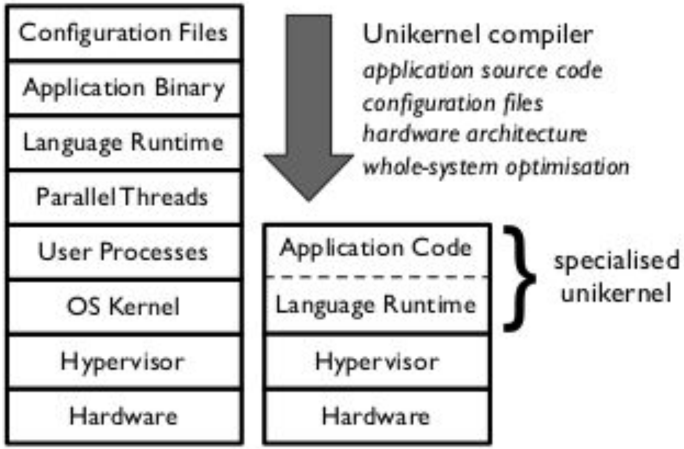
Source: tibco.com

Evolution of OS Environments

- Hypervisor
  - 10 machines running at 10% utilization, we just put them all on the same machine!
  - Hypervisors like Xen, Hyper-V, and KVM let us run isolated operating systems on the same hardware
  - Enables cloud computing on shared, but isolated resources
- Does it solve all our problems? No
  - Linux kernel is >25 million LOC (mostly C 😳), >100 syscalls, and endless interfaces
  - Giant attack surface
  - If we only want to run a single program, a traditional OS will have a ton of unnecessary overhead

Unikernels

- Custom kernels that run a **specific application**
- Write our operating system as a library, rather than a monolithic system
- Custom kernels import and link necessary interfaces
  - Networking, Graphics, Disk Blocks, Crypto, Entropy/Randomness, Time, DNS
- Compile down to a kernel with the **minimal set of features** needed to **run a specific program**



Source: Mirage.io

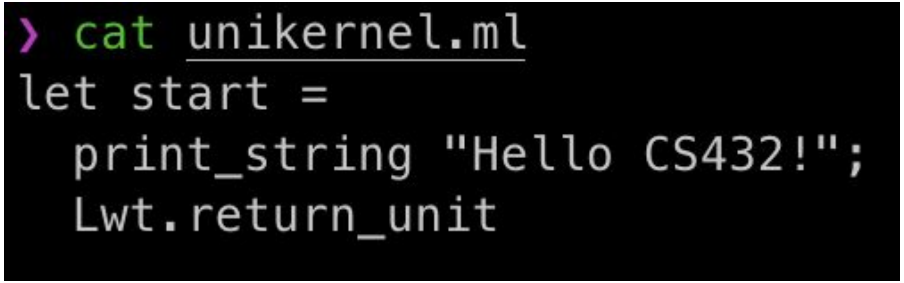
MirageOS

- Library operating system for constructing Unikernels
- Written in OCaml
    - Automatic Memory Management
    - Static type-checking and conducive to formal verification
    - Compilation to native code on most platforms
    - Powerful module system for organizing code
    - Worst-case usually within 2x C

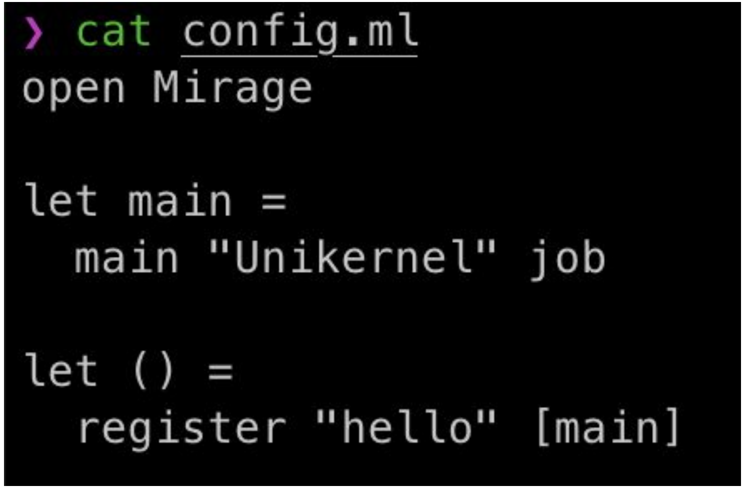
MirageOS: The Guts

- No concept of users, virtual memory, processes, scheduling, or privileges
- “Core” handles CPU + Memory
- Optional abstractions on top of this core
- Compiler can produce application code or compile down to a bootable OS
- Separation of interface signatures and implementation
  - Libraries can run on Unix during development and compile to OS drivers for production
- Event-driven
  - No preemptive threading: programs run until they explicitly pass off control
- First-class support for MacOS, Linux, BSD, Xen, KVM, and more

MirageOS: Writing a “Hello World” Kernel



```
❯ cat unikernel.ml
let start =
  print_string "Hello CS432!";
  Lwt.return_unit
```

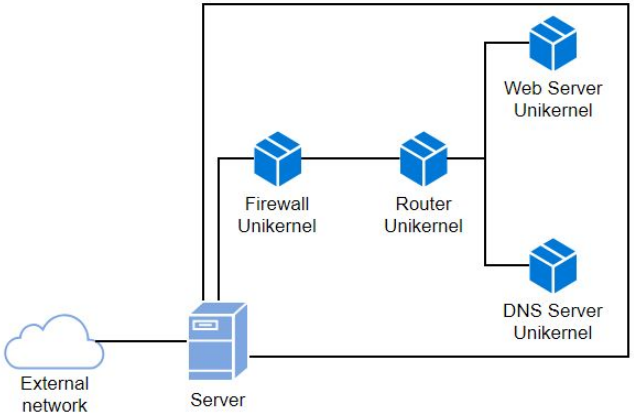


```
❯ cat config.ml
open Mirage

let main =
  main "Unikernel" job

let () =
  register "hello" [main]
```

MirageOS: Advantages



- Tiny Binaries (usually ~100-200kb)
- Tiny memory footprint (a few MB on average)
- Blazing fast startup times (20ms)
- **JIT operating systems**: Receive a query, boot a kernel, process the request, and send it back
- Self-scaling on-demand
- Cross-optimization of kernel and application code
- Eliminates many vulnerabilities (eg. buffer overflows)
- Possible to formally verify critical components

MirageOS: Disadvantages

- Terrible approach for traditional systems
- Programs must be written in pure OCaml
  - Technically possible to link C code, but arduous and potentially unsafe
- No support for protocols with closed specifications
- Illusion of security: Hypervisor vulnerabilities and unrestricted permissions

MirageOS: Further Resources

- https://mirage.io
- https://unikernel.org
- Other Unikernel Projects
  - HalVM (Haskell)
  - GuestVM (Java)
  - LING (Erlang)
  - IncludeOS (C++)
  - Clive (Go)
  - OSv (C, JVM, Ruby, Node.js)
  - Runtime.js (Javascript)
  - Rumprun (POSIX-compliant binaries)
  - Unik