H2020-ICT-2018-2-825377

UNICORE

UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments

Horizon 2020 - Research and Innovation Framework Programme

D2.4 API Design

Due date of deliverable: 31 December 2019

Actual submission date: 31 December 2019

Start date of project	1 January 2019
Duration	36 months
Lead contractor for this deliverable	University of Liège (ULiège)
Version	2.0
Confidentiality status	"Public"

Abstract

The goal of the EU-funded UNICORE project is to develop a common code-base and toolchain that will enable software developers to rapidly create secure, portable, scalable, high-performance solutions starting from existing applications. The key to this is to compile an application into very light-weight virtual machines – known as unikernels – where there is no traditional operating system, only the specific bits of operating system functionality that the application needs. The resulting unikernels can then be deployed and run on standard high-volume servers or cloud computing infrastructure.

In order to provide highest levels of flexibility during the Unikernels creation, it is necessary to define interfaces to interact with the internal components of the OS kernel. These interfaces will be used to expose existing OS kernel elements as a set of micro-libs (μ-libs). This decomposition of existing OS kernel components into μ-libs will define APIs to interact with Unikernels. *E.g.*, APIs to schedulers, timer facilities, memory management, network stacks, *etc*.

This deliverable will thus define library categories, and describe their associated APIs along with semantics annotations, in order to provide support for as many applications as possible. In addition, this deliverable will also identify the libraries that UNICORE will need to support the widest possible range of applications, along with a workplan as to how to quickly implement them. This deliverable is the second milestone in a series of three. Further improvements and information will be provided in the last version of this deliverable.

Target Audience

The target audience for this document is **public**.

Impressum

Full project title	UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments
Title of the workpackage	WP2 Platform design and evaluation
Editor	University of Liège (ULiège)
Project Co-ordinator	Emil Slusanschi, UPB
Technical Manager	Felipe Huici, NEC
Copyright notice	

Executive Summary

This is the second version of the UNICORE D2.4 document, ”API Design”.

This document uses D2.1 requirements as input in order to define UNICORE's APIs. The approach taken is to first describe the general structure of the UNICORE project as well as its main objectives. In order to respect those, the UNICORE project should provide an abstract layer to support decomposition and modularization of OS components into fine-grained modules called micro libraries or μ-libs. Such decomposition will allow developers to select and include only the μ-libs that are necessary for specific applications. This interface should also allow the construction of unikernels on several platforms and architectures.

Once the general structure of the UNICORE project has been defined, several operating system concepts have been described. These are not defined to fully explain the functions of an operating system, but rather to briefly describe the components that can be considered for API design. Taking into account the previously defined concepts as well as the requirements of the deliverable D2.1, it was possible to define several categories of API. It includes networking, storage, memory management, console (I/O), scheduler, thread management, time management and miscellaneous. Each category is described via high-level specifications and provides general operations to manipulate abstractions. These are needed not only to support the fundamental modularization of OS primitives in UNICORE, but also to facilitate unikernels and μ-libs verification.

Concerning semantic annotations, our approach is to verify each API exposed by the μ-libs. Each API is given a specification that is then checked against the implementation. Changing the implementation without changing the API (separation between policy - the API - and mechanism - the implementation) will not affect the specification. This connects the specification only to the API. For this goal, an automated tool (i.e. the verifier) will be implemented. The annotations will either be defined in separate locations or as code comments in the source code, ignored by the compiler. The verifier will parse both the annotations (specification) and the source code (implementation) and check the former against the latter.

By using these specifications, several μ-libs have already been defined. In order to provide better flexibility these have been divided again into three categories. The first category concerns internal libraries and provides functionality typically found in operating systems and are part of the UNICORE core. The second one is related to external libraries. These consist of existing software projects external to UNICORE but also language environments. UNICORE has one last type of micro-library: platform libraries. These libraries are the ones that allow to seamlessly support a range of different virtualization technologies *independently* of what the target application might be. In this way, UNICORE removes one of the big barriers of entry to adopting unikernel technologies: having to spend months of expert work building a unikernel for each potential virtualization technology has often proven to be a show stopper in terms of business adoption.

UNICORE has already provided several μ-libs that can be used with to build minimal multi-platforms/architectures unikernel. Other μ-libs are in progress or will be implemented for the next version of this document.

List of Authors

Authors	Gaulthier Gain and Cyril Soldani (ULiège), Felipe Huici (NEC), Razvan Deaconescu (UPB)
Participants	ULiège, NEC, UPB
Work-package	WP2 - Platform Design and Evaluation
Security	PUBLIC
Nature	R
Version	2.0
Total number of pages	37

Contents

List of Figures

List of Tables

1 Introduction

Operating systems expose different interfaces and features to applications to manipulate hardware. For example, a UNIX-like operating system allows to interact with the hardware by providing system interfaces such as a system call table. If such an interface does not exist, developers have to develop against hardware interfaces defined by manufacturers which will restrict applications to specific hardware. By providing such an interface, operating systems allow developers to program consistent applications regardless the underlying infrastructure.

Nowadays, the Linux kernel [1] has more than 400 different system calls [2]. Several system calls are very similar and can only be differentiated by their number of parameters. Securing such a syscall API is thus quite challenging. The advent of unikernels will circumvent this problematic since they include only the minimum functionalities to run a dedicated service. To provide such functionalities, a consistent system interface must be defined.

The challenge in this context is to define an abstraction layer allowing specialization and customization of unikernels while supporting multiple platforms (*e.g.*, bare metal, KVM [3], Xen [4], *etc.*) without requiring any additional work from the application developer.

Before defining the system interface itself, the main objectives to consider for developing a well-adapted interface are first explained in this chapter.

1.1 Objectives

The main objective of this deliverable is to provide the library categories API definitions and semantics annotations. This layer defines the common interfaces prevailing throughout UNICORE to support decomposition and modularization of OS components, and automated unikernels construction.

Defining a consistent system interface provides many advantages. Among these we can consider the following main concepts:

- *Simplicity of programming*: With APIs, developers can leverage the development by implementing the software that was already developed by other programmers. This way a programmer can fully focus on the core values of his application.

- *Separation of concerns*: The principles of separation of concerns allow to provide software modularity because it separates its code into different modules. Modularity, and therefore separation of concerns, is achieved by encapsulating information in a section of code that has a well-defined interface. Applying the separation of concerns principle simplifies the development and maintenance of computer components.

- *Ease of access*: Another advantage about APIs is that it makes a lot of great enhancements and features easily accessible. This fact is reinforced by the open-source nature of the project. With this approach, it

will be possible to have feedback about the interface and to take it into consideration to modify current components or add new features.

These different concepts will make it possible to define a programming interface proposing to choose the most adapted libraries to build a unikernel with optimal performance and tiny memory footprint. For example, a user can decide to use a very minimalistic libc implementation or a standard libc such as newlib [5] during the creation of a unikernel. By using the first one, the result image will be much lighter.

1.2 Organization

This document is organized as follows. Chapter 2 gives an overview of the UNICORE system, and discusses its general design principles. Chapter 3 describes the various components and their APIs. Chapter 4 describes the use of semantic annotations to maximize the correctness of API implementation. Chapter 5 gives an overview of existing μ-libs that have been designed for the Unikraft project and for future μ-libs. Finally, Chapter 6 summarises what has been achieved and any shortcomings that have been identified.

2 Design Principles

This chapter will first define the general principles of the UNICORE project. Then, it will cover general information that is relevant to all the operations specified in Chapter 3. These design principles will drive the UNICORE's API definition.

2.1 System overview

UNICORE will allow the decomposition of operating system primitives and libraries into fine-grained modules called micro-libraries or μ-libs. This decomposition of OS functions and system libraries into μ-libs will drive the definition of APIs. This set of APIs can be considered as an abstraction layer that will provide an interface to interact with lower levels. To turn an application into a unikernel, developers only have to follow the UNICORE API instead of implementing themselves the system API specification. In other words, instead of developing an application for each combination of target platform and architecture, they need only to use the defined APIs. Figure 2.1 illustrates a high level diagram of a unikernel on different platforms and architectures.

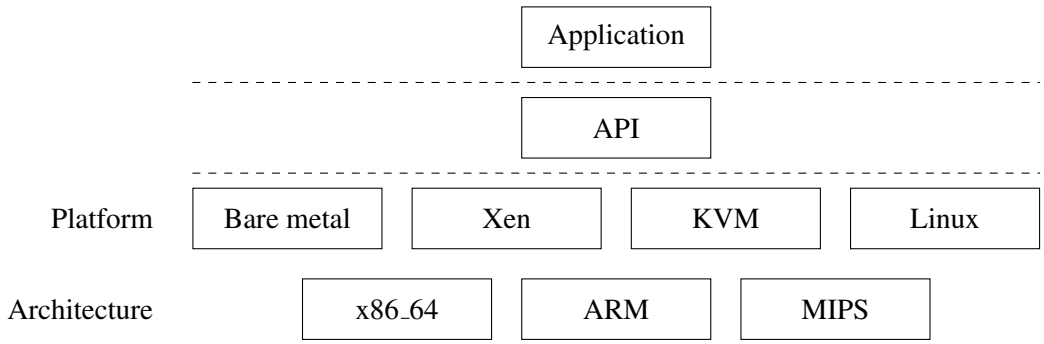

Figure 2.1: High-level overview of an application using the interface with different platforms and architectures.

As we can see this abstraction layer will allow to migrate an application on multiple platforms (*e.g.*, Xen and KVM) and CPU architectures. In addition to supporting several applications and architectures, UNICORE will offer several μ-libs. These μ-libs will provide a common code base for unikernels, ensuring a large degree of code reusability. Several μ-libs will be defined in order to guarantee as much modularity, flexibility and interoperability as possible.

In addition to propose several libraries, the UNICORE system will provide a build tool in charge of compiling the application and the selected libraries together to create a binary for a specific platform and architecture. The tool allows users to select libraries, to configure them, and to warn them when library dependencies are not met. In addition, the tool can also simultaneously generate binaries for multiple platforms. The build tool will be described in further details in deliverable D4.1 - Design & implementation of tools for unikernel deployment.

Figure 2.2 shows an example of a web service application packaged as a unikernel. This one is built by using

a special collection of needed μ-libs which can be chosen among a large set of μ-libs during the unikernel creation.

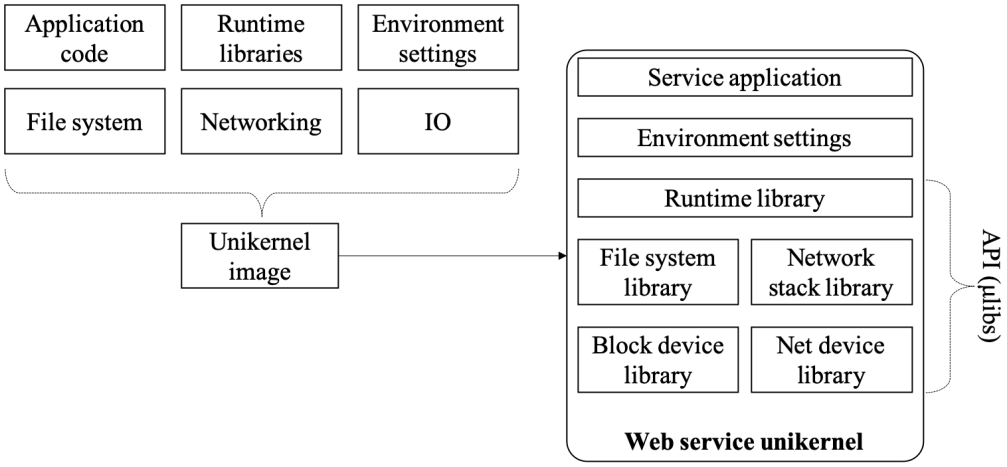

Figure 2.2: Web service unikernel built by the UNICORE system.

Finally, it should be noted that by combining the application, libraries and OS into a single entity, unikernels can avoid costly context switches and copies between kernel space and user space. Everything can be run in the same address space with the same privilege mode, increasing performance compared to traditional user-space applications running on top of a general-purpose operating system. Of course, there might still be copies and context switches in the underlying hypervisor (depending on the target platform), but at least redundant operations can be avoided.

2.2 Related work

There exist plenty of related work showing that unikernels bring great benefits and impressive performance compared to traditional VMs and containers. Broadly speaking, there exist two approaches to build unikernels:

(i) Development of minimalist operating systems that are POSIX-compliant. These can run existing and legacy application by using cross-compiling techniques. Generally, they are based on a custom kernel and use a larger code base since they require more resources. Nevertheless, these platforms provide an easier way to migrate traditional software (running on virtual ma- chines and containers) into unikernels, since the application only needs to be recompiled. OSv[6] is an example of this type of system. It is designed to run unmodified Linux applications on the KVM hypervisor. In the same family, Rumprun[7] provides reusable kernel-quality components which allow to build highly customized images with minimal footprint.

(ii) Development of minimalist operating systems with custom API. Unlike the previous approach, this model does not try to optimize existing code, but instead focus on a set of tools to quickly assemble new components without having to deal with underlying services (e.g., memory allocators, drivers,

...). The downside of this concept is that it provides code base which are generally incompatible with existing applications. Therefore, they require to rewrite the legacy code using the defined platform's API. For example, MirageOS[8] written in OCaml, is based on this architecture. It is designed as a complete clean-slate set of protocol libraries with which to build specialized unikernels that run over the Xen hypervisor.

UNICORE will rely on the first approach in order to be POSIX-compliant. Indeed, it must run legacy application by using cross-compiling techniques instead of being incompatible with existing applications.

2.3 General information

The purpose of this section is not to fully describe the functions of an operating system but rather to define and describe briefly the OS components which can be considered for designing the API in the following chapter.

2.3.1 I/O Streams

An I/O stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
In input operations, data bytes flow from an input source (such as keyboard, file, network or another program) into the application. In output operations, data bytes flow from the application to an output sink (such as console, file, network or another program). Streams acts as an intermediaries between the application and the actual I/O devices, in such a way that it frees the developer from having to handle the actual devices.
Unikernels will use I/O streams for a variety of operations such as sending data messages across the network or to write output to the console.

2.3.2 CPU Scheduling and processes

In UNIX systems, programs are run as processes. A process can be defined as a program instance that is running on one or more processor(s) under the control of an operating system. A process comprises a set of instructions for the processor, but also data that is stored in memory and a context. To manage processes, a classic operating system saves multiple pieces of information in data structures. There is a table to hold information about all created processes. There is one input per process in the table, called the Process Control Block (PCB). This structure is used to track the process's execution status and is thus needed to manage the scheduling of a particular process.
With unikernels, we have only one memory space and only one application that runs within. Therefore, managing several processes is no longer necessary. Section 3.5 discusses how to handle this new paradigm.

2.3.3 Threads

In unikernels, UNIX processes do not exist. Instead, the main structure is composed of threads. A thread is a flow of execution within a process which is characterized by a thread control block. As for processes, this

data structure will be used by the kernel's scheduler to determine which thread is allowed to execute at any point in time.

Nevertheless, unlike processes, threads can be easily ported on unikernels. Section 3.6 describes the high-level operations to manage threads.

3 API Design

This chapter provides high-level API specifications of the μ-libs. As explained previously, UNICORE should be POSIX-compliant [9] in order to run legacy application. In addition, to the POSIX specification, we define here high-level operations are mentioned. They represent general mechanisms to manipulate abstractions such as storage, networking, *etc.*, and will allow to drive the decomposition of OS and library primitives into μ-libs. This decomposition will be further detailed in Chapter 5 which presents some of the implemented μ-libs.

3.1 Objectives

In order to respect the requirements defined in the deliverable D2.1 [10], the UNICORE project shall meet the main following criteria:

- The system shall provide a high-level interface to interact with low-level features independently of the underlying platform and architecture.
- The system shall support all utilities and facilities related to Kubernetes [11] and OpenStack [12] orchestration.
- The system shall support DPDK [13] and NFV [14] infrastructures.
- The system shall support interactions with databases such as Redis [15] , MySQL [16] and SQLIte [17].
- The system shall support I/O interactions.
- The system shall support secure and insecure network connections.

Among all those criteria, it is possible to generalize the requirements by defining several categories. Having different categories allows to ensure a good cohesion and a separation of concerns while designing the APIs. The following categories have been defined:

- Networking.
- Storage.
- Memory management.
- Scheduler.
- Thread management.
- Console (I/O).
- Time management.
- Miscellaneous.

Designing a good API, in any language, is not easy. Arguably, the more abstraction the programming language allows, the better the APIs that can be created for it. According to this line of thought, designing a good API for low-level languages is particularly difficult, since they don't provide a lot of abstraction tools. In this

case, we define APIs by staying at a certain level of abstraction in order to provide underlying operations without having to know how they are implemented.

The design described in this document achieves elegance by partitioning system functions into eight major categories where each group of the system provides a well-defined service. Each category defines a set of concepts that can be exploited to perform particular tasks, as implied by its name. The following sections describe the API per category.

3.2 Networking API

Unikernels can be used for various purpose, and different applications will have different networking requirements. *E.g.*, a web server is an end-point application that will likely use a POSIX socket interface to abstract TCP connections as streams of bytes. On the other hand, a network address translator (NAT) is a middlebox that just processes packets, and does not need a TCP/IP stack at all.

In a general-purpose operating system such as GNU\Linux, those conflicting networking requirements are reconciled through the use of a very general and flexible networking stack, that can be used at the same time for various kinds of applications. However, that flexibility comes at a cost: *performance*. While performance is usually decent enough for end-point socket-based applications, it is not for middleboxes. Even if those are doing simple packet processing, they still have to pay the price of the full-blown networking stack, and the kernel-to-user-space copies. This hinders the development of network function virtualisation (NFV). To circumvent those problems, middleboxes have to find ways to bypass the OS network stack completely. This can be done through the use of user-space packet I/O frameworks, *e.g.*, the *Data Plane Development Kit* (DPDK [13]). Alternatively, one can also choose to integrate more closely into the kernel, *e.g.*, using eXpress Data Path (XDP [18]) which provides entry points at the lowest level in the network stack.

With unikernels, the situation is better. The custom-built network stack can be tailored to the application, and we only brings in what is strictly necessary. This improves both performance (as it incurs less overhead) and security (as it reduces attack surface). The various parameters tuning the networking stack can also be tailored to the unique application. Moreover, as far as the unikernel is concerned, there is not distinction between kernel space and user space, which can avoid costly context switches and packet copies (in some cases).

To ease the porting of existing network applications to the UNICORE platform, we will reuse existing networking APIs. For middlebox applications, which only process packets or are doing flow reconstruction themselves, we will provide raw packet I/O APIs, *e.g.* the one of DPDK. Conceptually, those APIs allow:

- to bind to and setup network interfaces;
- to allocate pools of packet buffers, used to receive and send packets;
- to receive and send batches of packets.

In combination with network virtualisation techniques such as SR-IOV [19], that allows to associate a physical NIC queue directly to a virtualised guest, this low-level interface should allow for very fast, lightweight middleboxes.

In addition to that low-level packet-based API, we will also provide a TCP/IP network stack for applications that need it, with a traditional POSIX socket interface.

Both the low-level and socket-based APIs will mimic existing APIs, to ease porting existing applications. However, we don't exclude the possibility of extending those APIs to take advantage of the specifics of the UNICORE environment.

Of course, following the general philosophy of UNICORE, the underlying implementations will be decomposed into different μ-libs, to maximise adaptability to the application and target environment (see section 5 for details).

3.3 Storage API

Storage API is designed to provide compatibility to existing applications, and flexibility to purpose built I/O intensive systems. The storage API comprises a file system and a block device layers. The file system layer offers the file and directory abstraction to applications, by leveraging interfaces exposed by the block device layer. The block device layer is directly facing storage device drivers. This layer is introduced for generality and flexibility to cope with various storage media and to build portable extensions. The extendability at the block layer allows developers to manipulate multiple storage hardware without changing interface for file systems. Typically this flexibility is useful to implement RAID and SSD cache. The following sections describe detailed specifications of storage related APIs.

3.3.1 File System Layer

POSIX compatible VFS interface. For backward compatibility, POSIX compatible functions are provided to applications. The interfaces rely on the VFS API which maintains file descriptors. Therefore, POSIX compatible interfaces have to be enabled under the dependency with the VFS layer. The table below shows some examples. Implementation of those functions are depending on file systems. Those interfaces can be explicitly obviated when building unikernels for specific-purpose applications.

Name	Description
`posix_open`	open a file and returns a file descriptor
`posix_close`	close a file
`posix_write`	write data to a file
`posix_read`	read data from a file

Table 3.1: POSIX compatible interfaces

File representation data structure. The main entity passed across the file system interfaces is the inode. Each file entity is corresponding to an inode object. To keep flexibility, definition of inode is kept minimal, and detailed data structures are defined in file system implementations. inode retains function pointers which

are a series of common file operations. Those functions are called from generic file operation routines. By implementing those function differently, file system specific behaviors can be realized. Table 3.2 shows some example of them.

Name	Description
`fgettype`	return the corresponding file type
`fgetname`	return the corresponding file name
`fwrite`	write data to the corresponding file
`fread`	read data from the corresponding file

Table 3.2: File operations

Directory operations. inode can also represent a directory. Directory inodes implement other functions which are specific to file and directory creation and deletion. File systems have to implement their own directory operations for the directory inode. When a file is created in a directory, those operations are executed.

Name	Description
`create`	create a file or directory
`delete`	delete a file or directory

Table 3.3: Directory operations

Interfaces for other functionalities. Here only defines minimal interfaces for hiding complexity of file system specific behaviors. In general, file system implementations incur high complexity in consistency management and every file system tends to be specialized for the crash consistency mechanism (*e.g.*, journaling, copy-on-write). Especially, data structure updates for block usage bitmap and directory trees are tightly coupled with file system specific crash consistency mechanisms. File systems have to explicitly call functions to indicate starting and ending transactions when they update their data structures, and such functions are too specific for file systems. Therefore, this storage API specification is designed to be as relaxed as possible so that developers can design and implement varied types of file systems. On the other hand, specifications in previous sections provide sufficient versatility for file systems so that application programs can run on different file systems without modifying their application code.

3.3.2 Block device interface

The block device layer is located in-between the file system layer and the storage device driver. This layer forwards requests from the upper layer to the lower layer. This layer is designed for adding flexibility to storage media management. The block device layer offers hook points to extensions so that they can implement storage array cooperation mechanisms such as RAID. The interface defined in Table 3.4 gives a representative example of this layer's interface. Those functions are called by an upper layer and take a block device representation object as an argument. The block device data structure implements also following functions, and at the bottom layer, device driver specific APIs are directly called. Unikernel applications can directly call those functions for bypassing the cost of file system operations.

Name	Description
block_write	write data to a specific block
block_read	read data from a specific block
block_write_hook	hook for block write
block_read_hook	hook point for block read

Table 3.4: Block API

3.4 Memory Management API

Unikernels are single-address space. As a result, there is no longer any separation between user and kernel address space. In this way, the kernel and the application can run in the same privilege ring. Although there is only a single-address space, memory management techniques are still needed.

Memory management can be divided into two parts. The first one concerns low-level operations and focuses on basics such as dynamic allocation of stack and heap storage while the second one provides high-level facilities.

(i) Low-level memory manager

- treats memory as a single, exhaustible resource

(ii) High-level memory manager

- manages pages within address space
- divides memory into abstract resources

3.4.1 Low-level interface

A set of functions and data structures should be used to manage free memory. Four high-level generic operations can be defined to manage memory:

Name	Description
allocstk	allocate stack space when a process is created
freestk	release a stack when a process terminates
allocheap	allocate heap storage on demand
freeheap	release heap storage as requested

Table 3.5: Low-level memory management operations

With this design, free space is considered as a single and exhaustible resource. The low-level memory manager allocates space provided a request can be satisfied. In addition, the low-level memory manager does not partition the free space into memory available for process stacks and memory available for heap variables. In other words, requests of one type can take the remaining free space.

Functions `allocstk` and `freestk` are respectively called during the process creation and termination. The `allocstk` obtains a block of memory from the highest address of free space, and returns a pointer to

the new allocated block. Finally, when the process is stopped or killed, a call to the function `freestk` is performed to release the process's stack and return the block to the free list.

Functions `allocheap` and `freeheap` provide similar services for heap management. Nevertheless, unlike the stack allocation functions, `allocheap` and `freeheap` allocate blocks from the lowest address of the free space.

3.4.2 High-level interface

The previous section describes a low-level memory management facility that treats memory as an exhaustible resource. At an upper level, conventional heap management operations can be defined. Indeed, it should be possible to perform classic operations to extend the heap memory by calling methods similar to `malloc`, `calloc` and `realloc` functions. The first method is used to dynamically allocate a single large block of memory with the specified size in bytes. If the space is insufficient, allocation fails and returns a NULL pointer. The second one is used to dynamically allocate the specified number of blocks of memory with zero values. Finally, the last one attempts to resize a memory block that was previously allocated. Basically, if tougher memory alignment is needed, it must be possible to use additional function(s) such as `memalign`. To deallocate a memory block previously allocated by a call to `calloc`, `malloc` or `realloc`, a `free` method taking as a parameter the pointer to a memory block previously allocated, should be defined.

Name	Description
`malloc`	allocate a single large block of memory
`calloc`	allocate the specified number of blocks of memory with zero values
`realloc`	resize a memory block that was previously allocated
`memalign`	allocate aligned memory
`free`	deallocate the memory previously allocated

Table 3.6: High-level memory management operations

3.5 Process API

A characteristic of a UNIX program is the use of multi-process abstractions, such as `fork` and interprocess communication. The use of such primitives allows multiple tasks to run independently of one another as though they each had the full memory of the machine to themselves. To exchange data and messages between processes, there exist several inter-process communication (IPC) mechanisms, each with unique use cases and semantics.

In the context of unikernels, the classical paradigm of processes should be adapted. Indeed, unikernels are dedicated to a single application and thus strip off the process abstraction from its monolithic appliance. In the same way inter-process communication can no longer be done using the traditional IPC.

As for classic operating systems, it should be possible to create a process with a specific function (*e.g.*, `fork` in UNIX). However in that case, this function should be used only once and during the unikernel start-up. Indeed, it will boostrap the process and thus the unikernel itself. It can be only used once and not

several times during the unikernel life cycle. Several UNIX applications use `fork` to handle simultaneously different operations. For example, mysql [16] uses `fork` when it is launched as a daemon. DNS servers such as rbldnsd [20] use `fork` for reloading DNS zones after having added new configuration. To handle existing applications that use `fork`, it is necessary to perform further research. When a unikernel must be stopped, a function to free up resources should be called. This function will stop the current process and thus the unikernel. Table 3.7 gives high-level operations to manage processes in a unikernel.

Name	Description
`process_create`	create a process as an unikernel instance
`process_exit`	terminate a process

Table 3.7: Process operations

Concerning the IPC mechanisms, those become useless since only one process is running at a time. Communications between unikernels should be handled by another mechanisms such as networking operations or/and Remote Procedure Call (RPC). The PCB data structure can be also modified since process scheduling is no longer needed. Again, further researches must be established concerning this part.

3.6 Thread Management API

Although unikernels only manage one process, they should have a complete support for SMP (multi-core) VMs, and for threads, as almost all modern applications use them. Thread API includes two abstractions: thread management (*e.g.*, creation of threads), and scheduling primitives for inter-thread synchronization and coordination.

3.6.1 Threads management

It should be feasible to easily create and exit threads with simple operations. Table 3.8 defines the high-level operations to manage threads. To ease the porting of existing multi-threaded applications to the UNICORE platform, we will reuse existing threads APIs such as pthreads [21] which is based on pre-emptive multi-threading.

Name	Description
`thread_create`	create and initialize a thread
`thread_exit`	terminate calling thread
`thread_join`	wait for a thread to finish
`thread_sleep`	suspend the current thread for the given timespan

Table 3.8: Basic operations for threads management.

The `thread_create` function starts a new thread in the calling process. The new thread starts execution by invoking an internal routine which defines its behaviour. Every thread created by `thread_create` should look identical to the platform and the architecture, to keep the abstraction portable on various host options. The purpose of `thread_exit` is to free the resources allocated for the current thread, including the initial stack. If several threads are created, it can be useful that the originating thread has to wait for the completion

of all its spawned thread's tasks. This operation is performed by calling the `thread_join` function. Finally, the `thread_sleep` method should suspend the current thread for a given timespan (in microseconds).

3.6.2 Threads synchronization and coordination

To coordinate and synchronize threads, the API should define two scheduling primitives. These scheduling primitives, listed in Table 3.9, should prevent a thread from spinning on a CPU core until the state of a lock or an event is atomically changed.

Name	Description
`mutex_lock`	Wait for a lock on a mutex object
`mutex_unlock`	Unlock a mutex object

Table 3.9: Basic operations for threads synchronization and coordination

The `mutex_lock` method sets up a mutex lock. A mutex lock enforces atomic execution in a critical section: it acts as a gate keeper to a section of code allowing one thread in and blocking access to all others until the lock is released again. The `mutex_lock` function releases a mutex lock held by the current thread.

To decide which thread should run, a thread scheduler should be implemented. The thread scheduler should be able to multiplex N threads on top of M CPUs (N may be much higher than M), and guarantee fairness and load balancing (moving threads between cores to improve global fairness). It should also consider pre-emptive or time slicing scheduling to schedule threads.

The approach describes below is called pre-emptive threading. Threads are pre-emptive, therefore if a given thread spends a lot of time using the CPU, the scheduler will temporarily interrupt it and will switch to another thread. There exists another model that is called collaborative threading. In this model, once a thread has control it executes until it explicitly yields control or blocks. The problem with cooperative multi-threading is that it depends on the willingness of threads to make room for other threads. If a task fails to release control, there is nothing that can be done about it.

A lot of web applications such as nginx [22] achieve concurrency with pre-emptive multitasking by using threads: each request is handled by a thread and when the request is done, the thread is released back to a pool. Some applications follow another approach by using the collaborative model. For example, the Ocsigen [23] framework allows to develop web sites and client-server web applications in OCaml [24] using a collaborative threading library called Lwt [25]. Depending the needs, it can be interesting to consider collaborative model and to provide a micro-lib based on the Lwt library.

3.7 Console API

Console API is designed to provide compatibility to existing applications by providing an interface to log messages and to retrieve user input. This one is especially useful for debugging purpose.

3.7.1 Output

On a standard Linux system, the user-space klogd daemon retrieves the kernel messages from the log buffer and feeds them into the system log file via the syslogd daemon. Considering unikernels, it is difficult to keep this approach since unikernels are designed specifically for running a single process. Therefore it should be necessary to propose several print operations and log levels. Although such an API is certainly less crucial than the one related to the network, it can be useful to debug applications. Table 3.10 shows the different log levels possible.

Name	Meaning
`LOG_EMERG`	Emergency messages, system is about to crash or is unstable
`LOG_ALERT`	Something bad happened and action must be taken immediately
`LOG_CRIT`	A critical condition occurred like a serious hardware/software failure
`LOG_ERR`	An error condition, often used by drivers to indicate difficulties with the hardware
`LOG_WARNING`	A warning, meaning nothing serious by itself but might indicate problems
`LOG_INFO`	Informational message *e.g.* startup information at driver initialization
`LOG_DEBUG`	Debug messages

Table 3.10: Log levels

In addition to log levels, several log operations should also be defined. These are mainly differentiated by their specific behavior. Indeed, the `log` method produces output with simple and standardized formatting. For more precise control over the output format than what is provided by `log`, the `logf` function should be used. With this one, it is possible to specify the width to use for each item, as well as various formatting choices for numbers.

Name	Description
`log`	write a message to the console
`logf`	write a message in a specific format to the console

Table 3.11: Log operations

In a general way, log methods write to standard error and print the date and time of each logged message. Every log message is output on a separate line: if the message being printed does not end in a newline, the logger will add one.

3.7.2 Input

Generally, unikernels do not interact directly with user input. This approach ensures better isolation and security. Nevertheless, in some cases (*e.g.*, debugging), it can be useful that users provide input. An `input` function allowing user input should also be considered. This one implements formatted input analogous to C's `scanf` function.

3.8 Time Management API

For several kinds of applications, time and date management can be useful. Therefore, it is necessary to define a group of functions to manage both. These functions should provide support for time acquisition,

conversion between date formats and formatted output to strings.

3.8.1 Date and time

It should be possible to return the current system time as the number of microseconds passed since the Epoch, 1970-01-01 00:00:00 Universal Time (UTC). To query the system time, the host must have a reliable time source. A common reliable time source that can be considered is a system timer incremented by the hardware alarm interrupts [26]. In addition, it should also be feasible to compute the difference in seconds between two time values.

Name	Description
`time`	return the current time of the system since an Epoch
`difftime`	compute the difference in seconds between two time values

Table 3.12: Time manipulation

3.8.2 Format conversion

The `time` function returns a value of integral type holding the number of seconds since the Epoch. Several functions to convert time since epoch to a (custom) textual representation should also be defined. For example, a textual representation can have the following format: Www Mmm dd hh:mm:ss yyyy

- Www - the day of the week (one of Mon, Tue, Wed, Thu, Fri, Sat, Sun).
- Mmm - the month (one of Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec).
- dd - the day of the month
- hh - hours
- mm - minutes
- ss - seconds
- yyyy - years

The `ttime` will convert a time value to the textual representation considering all fields below. Unlike the previous function, the `ctime` will allow to specify a specific string format.

Name	Description
`ttime`	convert a time value to a textual representation
`ctime`	convert a time value to a custom textual representation

Table 3.13: Format conversion

3.9 Miscellaneous API

Besides managing host resources, the API also contains miscellaneous features such as exception handling and random number generation.

3.9.1 Exception handling

Exceptions are designed for handling failures inside the exposed interface or for hardware exceptions. If an exception is triggered, a handler should be specified in order to catch it. When an exception is raised,

this handler should receive a data structure which contains information related to the exception. Among this information, it should be possible to get values of general-purpose registers when the exception is triggered, the address that triggers a memory fault, or an illegal instruction and an exception code. Table 3.14 contains a non-exhaustive list of possible exception codes:

Name	Description
MEM_FAULT	read or write memory violation
DIV_BY_ZERO	divide by zero
STACK_OVERFLOW	the stack went beyond the maximum available size
INST_FAULT	illegal instruction fault

Table 3.14: Possible exception codes

When an exception is raised, the current execution is interrupted and redirected to the specified handler function. This function will try to recover the execution by considering the previous information. Once the exception has been processed by the handler function, it can return to the original execution. Defining exception handling allows to support languages such as C++.

3.9.2 Random generator

Random numbers are used in various programs and protocols. Those are particularly useful for secure communications. Indeed, random numbers are one of the main elements for many cryptographic applications. Pseudo-random number generator (PRNG), encryption keys and authentication tokens need to be fed by random numbers. Standard functions defined in Table 3.15 respectively generate the pseudo-random number and initialize the seeds for the random number generator function.

Name	Description
rand	generate the pseudo-random number
srand	initialize the pseudo-random number generator

Table 3.15: Random generator operations

4 Semantic Annotations

Given the reduced code base (and, thus, attack surface) of the Unikraft μ-libs, we aim to ensure or maximize correctness of the implementation. Such that for any given input and environment configuration the behaviour will be consistent and as expected. Ensuring correctness requires, apart from the implementation of μ-libs, in specifications and a solution (prover) to verify the implementation against the specification [27]. The verifier will typically provide a positive, negative or uncertain response; in case of a negative response, the implementation is incorrect; in case of an uncertain response, the verifier is unable to track the entire implementation (such as because of infinite loops) or is using an incomplete verification.

Our approach is to verify each API exposed by the μ-libs. Each API is given a specification that is then checked against the implementation. Changing the implementation without changing the API (separation between policy - the API - and mechanism - the implementation) will not affect the specification. This connects the specification only to the API. It will be the developer or API designer that defines in the specification the expected behaviour of the implementation exposing the API. We will use an automated tool (i.e. the verifier) that is given as input the specification and the API implementation (within a μ-lib) and checks it.

We will define the specifications as semantic annotations to the each API. The annotations will either be defined in separate locations or as code comments in the source code, ignored by the compiler. The verifier will parse both the annotations (specification) and the source code (implementation) and check the former against the latter. The annotations will specify conditions under which the API implementation is expected to run and the expected behaviour and outcome. These conditions are defined formally and are parsed by the verifier. The verifiers, as common in verification, will use a typical SMT-solver approach to see if the conditions hold for the implementation.

A situation may arise where a negative response by the verifier is due to a faulty/incomplete specification. It will be the responsibility of the developer/specification writer to create proper annotations. When writing specifications, we typically aim for soundness and completeness. When we have sound specifications, we know that a correct response means the implementation is valid; if the response is incorrect, we can't tell; conversely, when we have complete specifications, we know that an incorrect response means that the implementation is invalid; if the response is correct, we can't tell. In practice, it's very difficult to reach soundness or completeness, not to say both. Specification writers will aim to maximize soundness and completeness (or one of them) of the specifications and thus maximize the correctness of the implementation.

We consider such an approach feasible due to recent works on verification of operating systems API (push-button verification) [28] [29]. The Yggdrasil system has been used for push-button verification of file systems and the continuing work has been used to verify a simplified UNIX-like kernel: `xv6`. The authors have updated the `xv6` kernel interface to allow automated reasoning; the resulting updated implementation was done using the Z3 SMT solver.

Another approach for operating systems verification is the one employed by the seL4 microkernel [30]. seL4

verification relied on constructing the entire specification for the implementation (not looking solely for the API), but the effort was substantial: 11 person years for 10,000 lines of C code.

Consequently, our approach is more similar to the one used in [29] relying on API specification and verification. This approach provides the benefit of modularity, targeting an API call at a time for verification and then moving from that; as a limitation, the approach will be unlikely to provide a sound and complete verification of each API implementation.

For specification writing and development of the verification solution we are considering HIP/SLEEK specifications [31]. HIP/SLEEK is a verification engine for heap manipulation programs, with some support for C programs. Incomplete support for C programs makes it unsuitable for our needs, but its specification format is a source of inspiration. HIP/SLEEK relies on pre-conditions (i.e. `require` rules) and post-conditions (i.e. `ensures` rules) to be added to blocks of code, such as functions (or μ-libs API in our case). Pre-conditions validate the initial conditions and API arguments; post-conditions validate the result and the outcome of the program.

In conclusion, with the addition of semantic annotations to μ-libs API we aim to ensure or maximize correctness of the implementation. The formal specifications will rely solely on the API definition; together with the implementation they will be fed to the verifier who will prove correctness or pin point the location for a verification error.

5 Micro-libraries from Unikraft

To allow for extreme specialization and customization of its unikernels, UNICORE will decompose operating system primitives and libraries into fine-grained modules called μ-libs. These can be arbitrarily small or as large as standard libraries like libc [32]. These libraries are divided into three categories:

(i) **Internal libraries** provide functionality typically found in operating systems and are part of the UNICORE core.

(ii) **External libraries** consist of existing software projects external to UNICORE. For example, these include libraries such as openssh [33], glibc and libuuid, but also language environments such as Javascript/v8 and Python [34].

(iii) **Platform libraries** allows users to select particular platforms.

In addition to these categories there exist also applications that can be ported to the UNICORE project. They correspond to standard applications such as MySQL, nginx or PyTorch [35], to name a few.
In the rest of this chapter we list the μ-libs that UNICORE has already provided, is working on providing, or is envisioning to provide in the future.

5.1 Internal Libraries

One important thing to point out regarding internal libraries is that for each category of library (*e.g.*, memory allocators, schedulers, device buses, network drivers, ...) UNICORE defines (or will define) an API that each library under that category must comply with. This is so that it is possible to easily plug and play different libraries of a certain type (*e.g.*, using a cooperative scheduler or a preemptive one).
An API consists of a header file defining the actual API as well as an implementation of some generic/compatibility functions, if any, that are common to all libraries under a specific category.

5.1.1 Networking and Communication

The network API provides information about the system's connection in terms of connection type (*e.g.*, wifi, wired, ...). It references also low-level functions to handle sockets. Indeed, the purpose of the networked-libraries is to manage several kinds of network applications.

- **uknetdev**: This library adds network driver interface. The netdev API provides a generalized interface between network device drivers and network stack implementations or low-level network applications.
- **ukbus**: API/abstraction for device buses
- **ukpci**: Implementation of a PCI bus (in progress)

5.1.2 Storage

- **vfscore**: The project's main virtual filesystem API and implementation
- **ramfs**: An implementation of a simple RAM-based filesytem
- **devfs**: An implementation of the device file system
- **9pfs**: An implementation of the 9pfs filesystem
- **libblock**: The project's block layer implementation and API

5.1.3 Memory Management

This subsection describes all project micro-libraries related to memory management.

- **ukalloc**: abstraction/API library for memory allocators
- **ukallocbbuddy**: binary buddy allocator implementation

5.1.4 Scheduling and Process

This subsection describes all project micro-libraries related to memory management.

- **uksched**: abstraction/API for all schedulers
- **ukschedcoop**: implementation of a cooperative round-robin scheduler
- **libukschedpreempt**: implementation of a pre-emptive scheduler (in progress)
- **libproc**: process support via cloning of running unikernel (in progress)
- **ukmpi**: inter-thread communication
- **uklock**: multi-task synchronization primitives

5.1.5 System and Miscellaneous

- **nolibc**: minimalistic implementation of libc functionality
- **noblim**: minimalistic implementation of libm functionality (in progress)
- **uktimeconv**: time conversion functions
- **ukboot**: early boot code
- **ukargparse**: argument parser
- **ukdebug**: printing and debug helpers
- **ukswrand**: random number generator

- **syscallshim**: shim layer for suppporting syscalls
- **uksglist**: singly-linked list implementataion
- **ukunistd**: implementation of unistd.h functions

5.2 External Libraries

In addition to internal libraries, UNICORE defines the concept of *external* micro-libraries. An external library adds functionality provided by a software package unrelated to UNICORE; for instance, these could include things like openssh, frameworks such as DPDK, or language environments such as Go [36] or Python [34]. External libraries can be added to a UNICORE unikernel build in order to enhance that unikernel's functionality. While clearly the external library's code doesn't need to be written from scratch, some level of porting effort needs to be done in order to integrate what is essentially an external project with the UNICORE build system. As complimentary work, we are looking at mechanisms to automatically port such libraries, for instance by using binary re-writing techniques to run Linux-built, ELF format binaries withing UNICORE-built unikernels.

The list below gives an overview of the external libraries currently supported by the project, as well as a few that are in progress.

(i) **newlib**: C standard library implementation

(ii) **musl**: C standard library implementation (in progress)

(iii) **libuv**: software library with a focus on asynchronous events (in progress)

(iv) **zlib**: software library used for data compression (in progress)

(v) **libuuid**: software library is used to generate unique identifiers

(vi) **libunwind**: library to analyze and modify the call stack of C programs.

(vii) **openssl**: software library for the Transport Layer Security (in progress)

(viii) **libaxtls**: software library for the Transport Layer Security (in progress)

(ix) **libcxx**: standard c++ library support

(x) **libcxxabi**: c++ abi

(xi) **compiler-rt**: runtime support

(xii) **eigen**: C++ template library for linear algebra

(xiii) **intel-intrinsics**: C style functions that provide access Intel instructions

(xiv) **fp16**: Half-precision floating point formats conversion

(xv) **fxdiv**: division via fixed-point multiplication by inverse

(xvi) **libunwind**: stack unwinder

(xvii) **lwip**: network stack

(xviii) **micropython**: sub-set of Python for embedded devices (in progress)

(xix) **pthread-embedded**: pthread API support

(xx) **pthreadpool**: pthread-based thread pool for C/C++

(xxi) **python**: CPython v2 and v3

(xxii) **libgo**: Go support

(xxiii) **libjvm**: OpenJDK/Java support (in progress)

(xxiv) **libruby**: Ruby support (in progress)

(xxv) **libv8**: Javascript/v8 support (in progress)

(xxvi) **librust**: Rust support (in progress)

(xxvii) **dpdk**: Intel's DPDK high performance packet framework (in progress)

(xxviii) **c-ares**: C library for asynchronous DNS requests

(xxix) **http-parser**: parser for HTTP messages written in C (in progress)

5.3 Platform Libraries

UNICORE has one last type of micro-library: platform libraries. These libraries are the ones that allow us to seamlessly support a range of different virtualization technologies *independently* of what the target application might be. In this way, UNICORE removes one of the big barriers of entry to adopting unikernel technologies: having to spend months of expert work building a unikernel for each potential virtualization technology has often proven to be a show stopper in terms of business adoption.

More concretely, UNICORE currently supports, or is planning on supporting, the following platforms:

- **Xen**: The popular open source Xen hypervisor is natively supported by UNICORE, including virtual drivers for networking and block devices (both in progress), as well as support for the Xen bus and XenStore.

- **KVM/QEMU** [37]: Support for the KVM platform, using QEMU for device emulation.

- **KVM/Solo5** [38]: Support for the minimalistic for highly efficient Solo5 virtual machine monitor (VMM) on top of KVM (in progress)
- **KVM/Firecracker** [39]: Support for the newly introduced Amazon Firecracker VMM (support for actual devices is in progress).
- **Linux userspace**: Not an actual virtualization platform, this target acts as a good development tool: UNICORE users can develop their unikernel using Linux user-space (and leveraging all the standard tools available in that environment), and then switch to one of the other platforms for actual deployment.
- **OCI containers** [40]: Support for OCI-compatible containers (*e.g.*, Docker and Rkt are OCI-compatible). This work is in progress.

In addition, it is worth pointing that UNICORE also supports multiple CPU architectures. Such architecture-specific code does not constitute actual micro-libraries, but does allow UNICORE to seamlessly support different architectures. So far, x86_64 is supported, with ARM64 support coming soon (there's also rudimentary support for ARM32). Support of the MIPS architecture should also be considered if we have enough resources and time.

6 Conclusion

This document describes the first milestone concerning design API. It begins by introducing the general structure of the UNICORE project as well as its main objectives. This one aims to define a clear and concise interface that can support multiple platforms/architectures and allows a certain degree of flexibility to the user by letting him to choose which library to pick to build an unikernel. Subsequently, it defines a set of concepts that is relevant to define general operations. All these specifications are established by the use of semantic annotations and define high-level mechanisms that should be considered during the design of a unikernel and more precisely concerning its interface. By using this specification, several μ-libs have already been defined and can thus be used with the build tool to construct a multi-platforms/architectures unikernel. Another μ-libs are in progress or will be implemented for the next version of this document.

References

[1] "The linux kernel archives." [Online]. Available: https://www.kernel.org

[2] "Linux system calls list." [Online]. Available: http://man7.org/linux/man-pages/man2/syscalls.2.html

[3] "Kernel virtual machine." [Online]. Available: https://www.linux-kvm.org/page/Main_Page

[4] "Xen project." [Online]. Available: https://www.xenproject.org

[5] "The newlib homepage - sourceware.org." [Online]. Available: https://sourceware.org/newlib

[6] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv—optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 61–72. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[7] "The rumprun unikernel and toolchain for various platforms." [Online]. Available: https://github.com/rumpkernel/rumprun

[8] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS'13*, vol. 48, no. 4, p. 461, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2451116.2451167

[9] "The open group base specifications issue 7, 2018 edition." [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/

[10] C. Patachia, E. Slusanschi, F. Huici, G. Bosson, G. Carrozzo, J. Martín, J. Bromell, J. Guijarro, M. Rapoport, R. Stoenescu, and R. Deaconescu, "D2.1 requirements," Apr. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2783992

[11] "Production-grade container orchestration." [Online]. Available: https://kubernetes.io

[12] "Build the future of open infrastructure." [Online]. Available: https://www.openstack.org

[13] "Dpdk - data plane development kit." [Online]. Available: https://www.dpdk.org

[14] "Etsi, european telecommunications standards institute, industry specification groups (isg) - nfv." [Online]. Available: https://www.etsi.org/technologies/nfv

[15] "Redis." [Online]. Available: https://redis.io

[16] "Mysql." [Online]. Available: https://www.mysql.com

[17] “Sqlite.” [Online]. Available: https://sqlite.org/index.html

[18] “Xdp - express data path.” [Online]. Available: https://www.iovisor.org/technology/xdp

[19] “Pci special interest group.” [Online]. Available: http://www.pcisig.com/home

[20] “A small and fast dns daemon especially made to serve dnsbl zones.” [Online]. Available: https://rbldnsd.io

[21] “Linux programmer’s manual - pthreads posix threads.” [Online]. Available: http://man7.org/linux/man-pages/man7/pthreads.7.html

[22] “Nginx — high performance load balancer, web server, and reverse proxy.” [Online]. Available: https://www.nginx.com

[23] “Multi-tier programming for web and mobile apps.” [Online]. Available: https://ocsigen.org/home/intro.html

[24] “Ocaml is an industrial strength programming language supporting functional, imperative and object-oriented styles.” [Online]. Available: https://ocaml.org/index.html

[25] J. Vouillon, “Lwt: A cooperative thread library,” in *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ser. ML ’08. New York, NY, USA: ACM, 2008, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1411304.1411307

[26] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[27] O. Demir, W. Xiong, F. Zaghloul, and J. Szefer, “Survey of approaches for security verification of hardware/software systems,” *IACR Cryptology ePrint Archive*, vol. 2016, p. 846, 2016.

[28] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, “Push-button verification of file systems via crash refinement,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’16. Berkeley, CA, USA: USENIX Association, 2016, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026879

[29] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, “Hyperkernel: Push-button verification of an os kernel,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17. New York, NY, USA: ACM, 2017, pp. 252–269. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132748

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “sel4: Formal verification of an os kernel,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems*

Principles, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629596

[31] W.-N. Chin, C. David, and C. Gherghina, "A hip and sleek verification system," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 9–10. [Online]. Available: http://doi.acm.org/10.1145/2048147.2048152

[32] "libc(7) - linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man7/libc.7.html

[33] "Openssh." [Online]. Available: https://www.openssh.com

[34] "Python." [Online]. Available: https://www.python.org

[35] "Pytorch - an open source deep learning platform that provides a seamless path from research prototyping to production deployment." [Online]. Available: https://pytorch.org

[36] "The go programming language." [Online]. Available: https://golang.org

[37] "Qemu, the fast! processor emulator." [Online]. Available: https://www.qemu.org

[38] "Solo5 unikernel." [Online]. Available: https://developer.ibm.com/open/projects/solo5-unikernel/

[39] "Firecracker - secure and fast microvms for serverless computing." [Online]. Available: https://firecracker-microvm.github.io

[40] "Open containers initiative." [Online]. Available: https://www.opencontainers.org

